LNCS 16220

Bernhard Steffen (Ed.)

Bridging the Gap Between AI and Reality

Third International Conference
on Bridging the Gap between AI and Reality, AISoLA 2025
Rhodes, Greece, November 1–5, 2025, Selected Papers

Springer

OPEN ACCESS

Lecture Notes in Computer Science 16220

The series Lecture Notes in Computer Science (LNCS), including its subseries Lecture Notes in Artificial Intelligence (LNAI) and Lecture Notes in Bioinformatics (LNBI), has established itself as a medium for the publication of new developments in computer science and information technology research, teaching, and education.

LNCS enjoys close cooperation with the computer science R & D community, the series counts many renowned academics among its volume editors and paper authors, and collaborates with prestigious societies. Its mission is to serve this international community by providing an invaluable service, mainly focused on the publication of conference and workshop proceedings and postproceedings. LNCS commenced publication in 1973.

Bernhard Steffen
Editor

Bridging the Gap Between AI and Reality

Third International Conference
on Bridging the Gap between AI and Reality, AISoLA 2025
Rhodes, Greece, November 1–5, 2025
Selected Papers

Springer

Editor
Bernhard Steffen
TU Dortmund University
Dortmund, Germany

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-032-07131-6 ISBN 978-3-032-07132-3 (eBook)
https://doi.org/10.1007/978-3-032-07132-3

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

As General and Program Chair I would like to welcome you to the proceedings of AISoLA 2025, the 3rd International Symposium on *Bridging the Gap between AI and Reality*, which took place on Rhodes (Greece) on November 1–5, 2025 as an in-person event, providing an interdisciplinary forum for discussing the impact of the recent AI developments in research, education, and society. It is our core belief that this topic must be explored from multiple perspectives to establish a holistic understanding. Therefore AISoLA invites researchers from various backgrounds, such as computer science, philosophy, psychology, law, economics, and social studies, to participate in an interdisciplinary exchange of ideas and to establish new collaborations. AISoLA is an AI-themed sibling of ISoLA, the International Symposium on Leveraging Applications of Formal Methods, which it complements with its interdisciplinary perspective.

The program of AISoLA 2025 consisted of three keynotes given by:

- Maximilian Kiener
- Edward A. Lee
- Alvaro Velasquez

And a collection of special tracks devoted to the following hot and emerging topics:

- AI Assisted Programming (AIAP) (Organizers: Wolfgang Ahrendt, Bernhard Aichernig, Klaus Havelund)
- Digital Humanities (DigHum) (Organizers: Ciara Breathnach, Tiziana Margaria, Tim Riswick)
- Formal Approaches in Intelligence for Transforming Healthcare (FAITH) (Organizers: Martin Leucker, Violet Kai Pun)
- Formal Methods for Intersymbolic AI (Organizers: Clemens Dubslaff, Ina Schaefer, Maurice ter Beek)
- Low Code/No Code Approaches to Application Developmemt: Challenges and Opportunities (Mike Hinchey, Tiziana Margaria)
- Responsible and Trusted AI: An Interdisciplinary Perspective (RTAI) (Organizers: Kevin Baum, Thorsten Helfer, Sophie Kerstan, Markus Langer, Eva Schmidt, Andreas Sesing-Wagenpfeil, Timo Speith)
- Small Data Challenges in AI for Material Science (Lars Kotthoff, Tiziana Margaria, Elena Raponi)
- Use of AI in the Industrial Sector (Falk Howar, Hardi Hungar, Barbara Steffen)
- 30 Years of UPPAAL (Kim G. Larsen, Paul Petterson, Wang Yi)

Co-located with the AISoLA Symposium were:

- The STRESS Summer School 2025 (Steve Bosselmann, Daniel Busch, Edward A. Lee, Bernhard Steffen)

The 14 papers of this volume represent a subset of the program of AISoLA 2025, most contributions will be published in the postproceedings.

We were pleased to implement a single-blind review process of all submitted content. Following the AISoLA tradition, the track organizers form the program committee. We thank them and the reviewers for their effort in selecting the papers to be presented. We are grateful to the Local Organizing Chair, Petros Stratis, and the EasyConferences team for their continuous precious support during the entire period preceding the events, and Springer for being, as usual, a very reliable partner for the proceedings production. Finally, we are thankful to Nicolas Stratis, Daniel Busch, and Steven Smyth for their continuous support of the website and the program, and to Steve Bosselmann for his help with the editorial system EquinOCS.

Special thanks are due to the Center for Trustworthy Data Science and Security, the Lamarr Institute for Machine Learning and Artificial Intelligence, and the Center for Perspicuous Computing, for their support in the organization of the event, as well as to the TU Dortmund, my home institution.

I hope all AISoLA participants had lively scientific discussions, ideally resulting in new collaborations and ideas that can be presented at next year's AISoLA, which will take place again in Greece.

August 2025 Bernhard Steffen

Organization

General Chair, Program Chair

Bernhard Steffen	TU Dortmund, Germany

Track Organizers/Program Committee

Wolfgang Ahrendt	Chalmers University of Technology, Sweden
Bernhard Aichernig	TU Graz, Austria
Kevin Baum	German Research Center for Artificial Intelligence, Germany
Maurice ter Beek	CNR, ISTI, Italy
Ciara Breathnach	University of Limerick, Ireland
Clemens Dubslaff	Eindhoven University of Technology, The Netherlands
Klaus Havelund	NASA Jet Propulsion Laboratory, USA
Thorsten Helfer	CISPA Helmholtz Center for Information Security, Germany
Mike Hinchey	University of Limerick, Ireland
Falk Howar	TU Dortmund, Germany
Hardi Hungar	German Aerospace Center and Carl von Ossietzky University Oldenburg, Germany
Sophie Kerstan	University of Freiburg, Germany
Lars Kotthoff	University of Wyoming, USA
Markus Langer	University of Freiburg, Germany
Kim G. Larsen	Aalborg University, Denmark
Martin Leucker	University of Lübeck, Germany
Tiziana Margaria	University of Limerick and Lero, Ireland
Paul Petterson	Mälardalen University, Sweden
Violet Kai Pun	Western Norway University of Applied Sciences, Norway
Elena Raponi	Leiden University, The Netherlands
Tim Riswick	Radboud University, The Netherlands
Eva Schmidt	TU Dortmund, Germany
Ina Schaefer	Karlsruhe Institute of Technology, Germany
Andreas Sesing-Wagenpfeil	Saarland University, Germany
Timo Speith	University of Bayreuth, Germany

Barbara Steffen	Janitza Electronics, Germany
Wang Yi	University of Uppsala, Sweden

Reviewers

Jonas Baumann
Thies de Graaff
Moez Ben Haihmida
David Hardin
Nicolas Hertzberg
Lennart Landt
Sara Mann
Julian Müller
Nadine Schlicker
Lena Schneider
Marjan Sirjani
Sarah Sterz
Per Strandberg
Jannik Zeiser
Thies de Graaff

Invited Keynote

Gaps in Generalization: Frontier Problems for Neurosymbolic AI

Alvaro Velasquez

University of Colorado, Boulder, CO, 80309 USA

Abstract. The Chat-GPT moment demonstrated that widely useful generalization is possible with AI. However, this capability is limited by classical assumptions on machine learning models, such as the test distribution matching the training distribution and the manifold hypothesis of relying on shared simple features across an otherwise complex dataset. These assumptions raise a critical question: how can AI generalize outside of such domains? Indeed, the foregoing assumptions are violated for important problems in autonomy, synthetic biology, and scientific discovery. We posit that, at some level of abstraction, the shared symbolic structures across domains will enable greater generalization and present research directions for neurosymbolic AI to achieve this vision of symbolic generalization that is robust to the gaps between AI and reality.

The field of neurosymbolic AI has witnessed a renaissance in recent years with the promise of achieving the best of the first two waves of AI. Whereas classical symbolic methods are performant for prescriptive problems in wellmodeled environments, they are not capable of handling noise and generalizing to unknown environments. On the other hand, modern deep learning architectures leverage neural networks to achieve some level of generalization to domains that match the training distribution of data, but they lack the interpretability and verifiability of the symbolic methods. In this paper, we present some open frontier problems as gaps that are unconventional and societally impactful and for which neurosymbolic AI is particularly well-suited to solve.

The Gap between Natural and Synthetic Biology

AI has largely influenced the biotechnology landscape by adapting techniques that have been successful with LLMs [1]. However, these systems reflect a limiting bias: the bias of nature itself. Indeed, large biological datasets reflect the proteins, DNA, and other processes from nature, which introduces a gap in the generalization of models when it comes to synthetic biology. Part of the problem stems from how these models are trained, with conventional pre-training over arbitrary sequences inducing a generalization with regard to sequence similarity. Consequently, synthetic biology applications remain difficult for such AI models since the synthetic sequence of interest may vary drastically in terms of sequence similarity to the training set of natural sequences. We therefore propose to leverage the symbolic structures that remain consistent across the natural and the synthetic: physics and logic. The former has been extensively studied in areas like molecular dynamics and the latter has been studied through the lens of formal languages. While conventional machine learning wisdom dictates that such symbolic structures would implicitly be learned with enough data, we instead argue for integrating their explicit representations in the form of, say, PDEs and context-free grammars, to improve data efficiency and generalization.

The Gap between Exploratory and Transformative Creativity

Imagine an AI whose capacity for creativity is so great that it can produce a body of work worthy of a Nobel prize in chemistry, physiology, or medicine. One can view this type of transformational creativity as an extreme form of out-of-distribution generalization. Intuitively, the more creative the ideas are, the less likely they are to be reflected in the training distribution of the underlying AI. However, there is an additional challenge for such an AI in that transformational ideas often challenge or invalidate existing knowledge. This challenge raises the question of how a creative AI can become so effective at generalization that it can ultimately invalidate its own training data. Although it seems like a contradiction in terms, a related type of reasoning is often referred to as infeasible or nonmonotonic reasoning, which can take on a symbolic form [2], thereby providing an avenue for the development of creative neurosymbolic AI for scientific discovery.

The Gap between Simulated and Real Autonomy

Real-world autonomy is limited by our inability to accurately model and simulate it [3]. However, otherwise different-looking environments may nevertheless share some underlying semantics. Thus, even though there are large sim-to-real gaps, the semantic gap in terms of some shared semantic abstraction is low. We believe that finding that level of abstraction where semantic similarity is high will provide the features along which robust generalization is possible. We argue that more research is needed on how to extract and leverage the explicit semantic abstractions that minimize the semantic gap between these environments as opposed to attempting to implicitly minimize the sim-to-real gap, which can be ineradicable in some cases. These explicit symbolic representations can be used to define novel neurosymbolic AI for robust autonomy by incorporating diverse environments and their shared semantic abstractions into the ML pipeline.

References

1. Wang, M., Zhang, Z., Bedi, A.S., Velasquez, A., et al.: A call for built-in biosecurity safeguards for generative ai tools. Nat. Biotechnol. **43**(6), 845–847 (2025)
2. Schapiro, S., Black, J., Varshney, L.R.: Transformational creativity in science: a graphical theory. arXiv preprint arXiv:2504.18687 (2025)

3. Noorani, E., Serlin, Z., Price, B., Velasque, A.: From abstraction to reality: Darpa's vision for robust sim-to-real autonomy. AI Mag. **46**(2) (2025)

Introduction to Section Formal Methods for Intersymbolic AI

Formal Methods for Intersymbolic AI

Maurice H. ter Beek[1], Clemens Dubslaff[2,3], and Ina Schaefer[4]

[1] CNR–ISTI, Pisa, Italy
m.terbeek@isti.cnr.it
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
c.dubslaff@tue.nl
[3] Centre for Tactile Internet with Human-in-the-loop (CeTI), Dresden, Germany
[4] KIT, Karlsruhe, Germany
ina.schaefer@kit.edu

Abstract. A key benefit of symbolic (rule-based) artificial intelligence (AI) is its formal rigor, which comes at the cost of formal modeling effort and computational expensive reasoning. Differently, subsymbolic (datadriven) AI approaches usually outperform rigorous ones in performance but might lead to unsound results. *Intersymbolic* AI is an emerging field in AI that aims to combine symbolic and subsymbolic AI approaches, exploiting the benefits from both worlds. The scope of the ISoLA 2025 track on "Formal Methods for Intersymbolic AI" is to gather researchers and practitioners from formal methods and (sub)symbolic AI to establish the boundaries of intersymbolic AI and to investigate and clarify the role of formal methods therein.

Keywords: Formal Methods for AI, AI-enabled Verification, Explainable AI

Motivation

In his keynote contribution during last year's ISoLA [4], Platzer [9] called for the study of the field that was coined *intersymbolic artificial intelligence (AI)*. This field targets the combination of *symbolic AI*, whose building blocks have inherent significance/meaning, with *subsymbolic AI*, whose entirety creates significance/effect despite the fact that individual building blocks escape meaning. Symbolic AI, as implemented in rule-based systems, provides formal rigor, but this comes at the cost of increased modeling effort and computationally expensive reasoning. Differently, subsymbolic AI approaches, which typically use data-driven methods from statistical learning, are not as computationally expensive as the rigorous ones but might lead to unsound results. The idea is that intersymbolic AI combines benefits from both symbolic and subsymbolic AI to increase the overall effectiveness, rigor, and explainability of AI compared to either kind of (sub)symbolic AI alone.

Instances of such combinations have already been established in the literature and showcase the broad applicability of the intersymbolic AI concept. The probably most established instance of intersymbolic AI is in *neurosymbolic AI*, which focuses on the combination of symbolic and neural network reasoning [1,10]. Other instances have also been reflected in last year's ISoLA track on "X-by-Construction Meets AI", where several contributions involved intersymbolic AI. For example, an *intersymbolic programming language* was proposed, pairing logical primitives with training and prediction based on subsymbolic methods [5]. Within the area of *explainable AI* (XAI), an incarnation of intersymbolic AI by means of *logic-based XAI* was addressed [7]. In this strand of XAI, symbolic AI by means of logic reasoning is used to explain classifiers learned using subsymbolic AI approaches. Such approaches are particularly important since the operation of the most advanced AI models is often beyond the grasp of human decision makers. Much work on XAI relies on measures to quantify feature importance such as SHAP [6]. While such measures can give an indication of which are the relevant aspects in AI components, they cannot rigorously explain them. In high-risk or safety-critical domains, more formal approaches at different levels of abstraction are required to build the much needed trust [8].

In formal methods, explainability is an ongoing topic of research, turning formal correctness proofs on decision-making processes into interpretable explications [2]. Subsymbolic approaches such as reinforcement learning may assume an underlying formal model, e.g., by means of a Markov decision process (MDP). These models can be subject to formal methods, such as probabilistic model checking or explainability through formal notions of causality [3].

Research Questions

All of the above mentioned areas only provide a glimpse of the many research questions and research opportunities that are emerging from the combination of formal methods and AI, constituted in the field of intersymbolic AI: What is the role of formal methods in intersymbolic AI? How can formal methods ensure rigorous explanations of intersymbolic AI approaches? Is there a generic methodology for intersymbolic AI that provides the benefits from both symbolic and subsymbolic AI approaches? What are the lessons learned from applying formal methods for neurosymbolic AI or other forms of intersymbolic AI?

Track Format

The track on "Formal Methods for Intersymbolic AI" (FMIAI) at AISoLA 2025 is organized as a two-day event to foster collaboration and research in intersymbolic AI. It addresses researchers and practitioners from formal methods, symbolic or subsymbolic AI, and established fields of intersymbolic AI such as neurosymbolic AI and XAI. Topics accepted for presentation range from logicbased XAI, verification and explanation of neural networks, and combinations of statistical model checking and reinforcement learning, to large language and modal models in a variety of application domains.

Track participants have the opportunity to extend their presented contributions and include aspects discussed during the conference, and submit their work for publication in the forthcoming post-proceedings volume of AISoLA 2025.

References

1. Bader, S., Hitzler, P.: Dimensions of Neural-symbolic Integration - A Structured Survey. arXiv:cs/0511042 [cs.AI] (2005).https://doi.org/10.48550/arXiv.cs/0511042
2. Baier, C., et al.: From verification to causality-based explications. In: Bansal, N., Merelli, E., Worrell, J. (eds.) Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP'21). LIPIcs, vol. 198, pp. 1:1–1:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPICS.ICALP.2021.1
3. Baier, C., Piribauer, J., Ziemek, R.: Foundations of probability-raising causality in Markov decision processes. Log. Methods Comput. Sci. **20**(1), 4:1–4:66 (2024). https://doi.org/10.46298/LMCS-20(1:4)2024
4. ter Beek, M.H., Cleophas, L., Dubslaff, C., Schaefer, I.: X-by-construction meets AI. In: Margaria, T., Steffen, B. (eds.) Proceedings of the 12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Software Engineering Methodologies (ISoLA'24). LNCS, vol. 15222, pp. 155–161. Springer (2024). https://doi.org/10.1007/978-3-031-75387-9
5. Belmonte, G., Bussi, L., Ciancia, V., Latella, D., Massink, M.: Towards hybrid-AI in imaging using VoxLogicA. In: Margaria, T., Steffen, B. (eds.) Proceedings of the 12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Software Engineering Methodologies (ISoLA'24). LNCS, vol. 15222, pp. 205–221. Springer (2024). https://doi.org/10.1007/978-3-031-75387-9
6. Lundberg, S.M., Lee, S.: A Unified approach to interpreting model predictions. In: Guyon, I., et al. (eds.) Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17). pp. 4768–4777. Curran Associates Inc. (2017). https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html
7. Marques-Silva, J.: Logic-based explainability: past, present & future. In: Margaria, T., Steffen, B. (eds.) Proceedings of the 12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Software Engineering Methodologies (ISoLA'24). LNCS, vol. 15222, pp. 181–204. Springer (2024). https://doi.org/10.1007/978-3-031-75387-9 12
8. Mienye, I.D., et al.: A survey of explainable artificial intelligence in healthcare: Concepts, applications, and challenges. Inform. Med. Unlocked **51**, 101587 (2024). https://doi.org/10.1016/j.imu.2024.101587
9. Platzer, A.: Intersymbolic AI: Interlinking symbolic AI and subsymbolic AI. In: Margaria, T., Steffen, B. (eds.) Proceedings of the 12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Software

Engineering Methodologies (ISoLA'24). LNCS, vol. 15222, pp. 162–180. Springer (2024). https://doi.org/10.1007/978-3-031-75387-9
10. Sheth, A.P., Roy, K., Gaur, M.: Neurosymbolic artificial intelligence (why, what, and how). IEEE Intell. Syst. **38**(3), 56–62 (2023). https://doi.org/10.1109/MIS.2023.3268724

Contents

Invited Keynotes

Extended Abstract: Will Embodied AI Become Sentient?

Edward A. Lee(✉)

University of California, Berkeley, USA
eal@berkeley.edu

Abstract. This extended abstract outlines technical reasons that embodiment qualitatively changes the nature of AI agents and potentially enables sentience. Specifically, I argue that knowledge can be purely subjective, not externally observable, and that sentience is this form of knowledge. I further argue that first-person interaction can gain knowledge that no objective observation can gain. And finally, I argue that the introduction of feedback through the physical world enables distinguishing self from non-self, an essential distinction for sentience. Putting all these together, I conclude that embodied AI agents may in fact become sentient, but also that we can never know for sure whether this has happened.

1 Embodied AI

I have previously argued [5,6] that the deep neural networks that underlie today's large-language models (LLMs) more closely resemble Kahneman's "system 1," the quick intuitive thinking that is not subject to conscious control, than "system 2," the slow, deliberate, and controlled thinking that leverages rationality and logic [4]. Like humans, they have "bounded rationality" [13], i.e., limited ability to reason logically and limited short-term memory. As a consequence, their real strengths are not in logical reasoning, but in pattern recognition and prediction.

Embodied cognition is a multidisciplinary thesis that argues that the mind is inseparable from the body [1,14]. Specifically, it argues that the mind is not a separate entity, but rather an emergent property of the body and its interaction with the environment. This is a radical departure from the traditional view of the mind as an objective property of the brain, a computation going on in the skull. Instead, the mind *is* the interaction of the body, including the brain, with its environment.

Sentience is the ability to experience feelings and sensations. It is valenced experience, experience that is positive or negative, good or bad. Sentience is subjective, not objective. It is not observable from the outside, measurable, or quantifiable. Sentience depends on embodiment, not just observation.

Today's LLMs have relatively little interaction with the physical world. They mostly interact with humans through the internet, but lack sensors and actuators that directly interact with their physical environment. But this is changing as AI technology is integrated into physical robots. Mon-Williams et al. [9]

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 3–7, 2026.
https://doi.org/10.1007/978-3-032-07132-3_1

have shown that embodied large language models can complete complex tasks in unpredictable environments. Here, I go further and explore whether such embodiment can lead to sentience.

Embodiment is not just about sensors and actuators. It requires bidirectional feedback. The sentient entity not only reacts to sensations, but also acts to sense. A neural network can learn to correlate its physical actions with consequent sensor inputs, thereby gaining the ability to distinguish self from non-self. Sensory input that depends on actuator outputs is not the same as sensory input coming from the non-self environment. Sentience requires such a distinction.

Here, I give three technical reasons that embodiment qualitatively changes the nature of AI agents and potentially enables sentience. These technical reasons are explained in much more detail in chapters 10–12 of my *Coeveolution* book [7]. First, I argue that knowledge can be purely subjective, not externally observable. Second, I argue that first-person interaction can gain knowledge that no objective observation can gain. Third, I argue that the introduction of feedback through the physical world enables distinguishing self from non-self, an essential distinction for sentience. Putting all these together, I conclude that embodied AI agents may in fact become sentient, but I also give technical reasons that we can never know for sure whether this has happened.

2 Subjective Knowledge

Dodig-Crnkovic [2] argues for a relational epistemology, where knowledge is not objective, but rather a function of the observer's relationship to the observed. This is a radical departure from the traditional scientific goal of knowledge acquired through purely objective observation. Can this be true? Can knowledge exist that is purely subjective, not externally observable?

A rather technical Turing-award winning result, zero-knowledge proofs [3], shows definitively that knowledge can be purely subjective, not externally observable. Zero-knowledge proofs provide a procedure for proving a statement without giving the recipient of the proof the ability to prove the same statement. Knowledge of the truth of the statement becomes subjective, held only by the recipient of the proof. For a delightful explanation of this result, see Quisquater et al. [12].

An interesting property of zero-knowledge proofs is that they do not really give definitive proofs, in the sense of formal logic, but rather give evidence of the truth of the statement. The recipient of the proof can, through interaction, build an arbitrary degree of confidence in the truth of the statement, short of absolute certainty.

Sentience is subjective knowledge. Zero-knowledge proofs provide a formal argument that knowledge can be purely subjective, not externally observable, so we don't have to fall back on non-technical arguments, intuition, or spiritual beliefs.

3 First-Person Interaction

Milner [8] shows that two state machines can be indistinguishable from each other by any objective observer, and yet exhibit different behavior when interacted with. In particular, two machines that "simulate" each other will look identical to any objective observer, but can exhibit different behavior when interacted with. Milner introduced a "bisimulation" relation as a stronger form of equivalence than mutual simulation. If two machines are bisimilar, then no observation *or interaction* can distinguish them; if they are only mutually similar, then no observation can distinguish them, but they may be distinguishable through interaction. This shows that first-person interaction can gain knowledge that no objective observation can gain. The first-person observer can see the difference between the two machines, but the objective observer cannot.

Sentience, similarly, is not observable from the outside. Humans, by interacting with one another, however, obtain evidence of sentience in each other. Empathy arises from such first-person interaction, which explains why humans find it more difficult to feel empathy for remote others.

The "first person" aspect of this interaction is essential. A first-person interaction depends on the observer's ability to distinguish self from non-self, which brings us to the final technical reason that embodiment enables sentience.

4 Feedback

The introduction of feedback through the physical world enables distinguishing self from environment. An agent that acts on the physical world and perceives through its sensors effects from that action can correlate its actions with its sensor inputs, thereby gaining the ability to distinguish self from non-self.

In an animal, a motor efference is a signal sent to the muscles to act. An efference feedback mechanism has been identified in even the most primitive animal nervous systems, where the motor efference is fed back into the sensory system, which learns to predict the sensations that will result from the action. Deviations from the predicted sensations are used to adjust the action, but also to identify components in the sensation that arise from the environment rather than the self. This provides even the most primitive organisms the ability to distinguish self from non-self. Sentience requires this ability to distinguish self from non-self. For an experience to have valence, there must be a self for whom the experience is positive or negative.

In more complex organisms, such as humans, efference feedback gives rise to the ability to reason about causation. The organism learns that its actions have effects, and that these effects can be predicted. This allows the organism to reason about some of the causes of its sensations, specifically those that are caused by its own actions.

Pearl [10,11], another Turing-award winner, argues that it is impossible to reason about causation objectively. He uses statistical methods to show that the only way to reason about causation is to use a causal model, which is a presupposed model of the causal relationships between the variables in the system. This model is not objective, but rather a subjective representation of the observer's understanding of the system. This is the basis of the ability to reason about causation.

Once sensations become valenced, ethical concerns arise. If the machines acquire sentience, then it may become reasonable to hold them accountable for their actions. It may also become important to give their "feelings" some consideration.

5 Conclusion

Putting all these together, I conclude that embodied AI agents may in fact become sentient, but also that we can never know for sure whether this has happened. No objective observation will be sufficient. Subjective interaction may give evidence of sentience, but doubts will remain. As with zero-knowledge proofs, subjective, first-person interaction may build confidence, but not certainty.

References

1. Clark, A.: Supersizing the Mind: Embodiment, Action, and Cognitive Extension. Oxford University Press, Oxford (2008)
2. Dodig-Crnkovic, G.: Observers as agents: Relational epistemology from physics to ecology. Qeios (2025). https://doi.org/10.32388/TLFZ3V
3. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM J. Comput. **18**(1), 186–208 (1989). https://doi.org/10.1137/0218012
4. Kahneman, D.: Thinking Fast and Slow. Farrar, Straus and Giroux, New York (2011)
5. Lee, E.A.: What can deep neural networks teach us about embodied bounded rationality. Front. Psychol. **25** (2022). https://doi.org/10.3389/fpsyg.2022.761808
6. Lee, E.A.: Certainty vs. intelligence. In: AISoLA: Bridging the Gap Between AI and Reality. LNCS, vol. 15217. Springer (2024). https://doi.org/10.1007/978-3-031-75434-0_2
7. Lee, E.A.: The Coevolution: The Entwined Futures of Humans and Machines. MIT Press, Cambridge (2020). https://ptolemy.berkeley.edu/~eal/books/TheCoevolution.pdf
8. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
9. Mon-Williams, R., Li, G., Long, R., Du, W., Lucas, C.G.: Embodied large language models enable robots to complete complex tasks in unpredictable environments. Nature Mach. Intell. **7**, 592–601 (2025). https://doi.org/10.1038/s42256-025-01005-x
10. Pearl, J.: Causality: Models, Reasoning, and Inference, 2nd edn. Cambridge University Press, Cambridge (2009)
11. Pearl, J., Mackenzie, D.: The Book of Why: The New Science of Cause and Effect. Basic Books, New York (2018)

12. Quisquater, J.-J., et al.: How to explain zero-knowledge protocols to your children. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 628–631. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_60
13. Simon, H.A.: Bounded rationality in social science: today and tomorrow. Mind Soc. **1**, 25–39 (2000)
14. Thelen, E.: Grounded in the world: developmental origins of the embodied mind. Infancy **1**(1), 3–28 (2000)

AI Assisted Programming

AI Assisted Programming (AISoLA 2025 Track Introduction)

Wolfgang Ahrendt[1,2], Bernhard K. Aichernig[3], and Klaus Havelund[4(✉)]

[1] Chalmers University of Technology, Gothenburg, Sweden
ahrendt@chalmers.se
[2] University of Gothenburg, Gothenburg, Sweden
[3] Institute for Formal Models and Verification, Johannes Kepler University Linz, Linz, Austria
bernhard.aichernig@jku.at
[4] NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena, 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 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.

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 11–17, 2026.
https://doi.org/10.1007/978-3-032-07132-3_2

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 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 [3–8] 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, automated 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. Ahrendt, W., Aichernig, B., Havelund, K.: AI assisted programming. In: Steffen, B. (ed.) AISoLA 2024 - Bridging the Gap Between AI and Reality. LNCS, vol. 15217, pp. 101–106. Springer (2025)
2. Ahrendt, W., Havelund, K.: AI assisted programming. In: Steffen, B. (ed.) AISoLA 2023 - Bridging the Gap Between AI and Reality, vol. 14380, pp. 351–354. LNCS. Springer (2024)

3. Etemadi, K., Sirjani, M., Moghadam, M.H., Strandberg, P., Pettersson, P.: LLM-based property-based test generation for guardrailing cyber-physical systems. In: Steffen, B. (ed.) AISoLA 2025. LNCS, vol. 16220, pp. xx–yy. Springer, Cham (2025)
4. Hajhmida, M.B., Lee, E.A.: RAg and agentic assistant: a combined approach. In: Steffen, B. (ed.) AISoLA 2025. LNCS, vol. 16220, pp. xx–yy. Springer, Cham (2025)
5. Hertzberg, N., Sevenhuijsen, M., Kåreborn, L., Lokrantz, A.: CASP: an evaluation dataset for formal verification of C code. In: Steffen, B. (ed.) AISoLA 2025. LNCS, vol. 16220, pp. xx–yy. Springer, Cham (2025)
6. Landt, L., Leucker, M., Burchardt, C.: A voice-enabled query framework for systems engineering artefacts. In: Steffen, B. (ed.) AISoLA 2025. LNCS, vol. 16220, pp. xx–yy. Springer, Cham (2025)
7. Müller, J., de Graaff, T., Möhlmann, E.: Integrating LLMs with QC-OpenDRIVE: ensuring normative correctness in autonomous driving scenarios. In: Steffen, B. (ed.) AISoLA 2025. LNCS, vol. 16220, pp. xx–yy. Springer, Cham (2025)
8. Tahat, A., Amundson, I., Hardin, D., Cofer, D.: AGREE-Dog Copilot: a neuro-symbolic approach to enhanced model-based systems engineering. In: Steffen, B. (ed.) AISoLA 2025. LNCS, vol. 16220, pp. xx–yy. Springer, Cham (2025)
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

LLM-Based Property-Based Test Generation for Guardrailing Cyber-Physical Systems

Khashayar Etemadi[1,2](✉), Marjan Sirjani[1], Mahshid Helali Moghadam[3], Per Strandberg[4], and Paul Pettersson[1]

[1] Mälardalens University, Västerås, Sweden
{marjan.sirjani,paul.pettersson}@mdu.se
[2] ETH Zurich, Zurich, Switzerland
ketemadi@ethz.ch
[3] Cloud and Embedded Platform, Traton AB, Södertälje, Sweden
mahshid.helali.moghadam@scania.com
[4] Westermo Network Technologies AB, Västerås, Sweden
per.strandberg@westermo.com

Abstract. Cyber-physical systems (CPSs) are complex systems that integrate physical, computational, and communication subsystems. The heterogeneous nature of these systems makes their safety assurance challenging. In this paper, we propose a novel automated approach for guardrailing cyber-physical systems using property-based tests (PBTs) generated by Large Language Models (LLMs). Our approach employs an LLM to extract properties from the code and documentation of CPSs. Next, we use the LLM to generate PBTs that verify the extracted properties on the CPS. The generated PBTs have two uses. First, they are used to test the CPS before it is deployed, i.e., at design time. Secondly, these PBTs can be used after deployment, i.e., at run time, to monitor the behavior of the system and guardrail it against unsafe states. We implement our approach in ChekProp and conduct preliminary experiments to evaluate the generated PBTs in terms of their relevance (how well they match manually crafted properties), executability (how many run with minimal manual modification), and effectiveness (coverage of the input space partitions). The results of our experiments and evaluation demonstrate a promising path forward for creating guardrails for CPSs using LLM-generated property-based tests.

Keywords: Property-based Testing · LLM4SE · Cyber-Physical System · Safety

1 Introduction

Cyber-physical systems (CPS) are integrated hardware-software systems where computation and physical processes are deeply intertwined. Ensuring safety [8] in

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 18–46, 2026.
https://doi.org/10.1007/978-3-032-07132-3_3

these systems, in particular for the safety-critical ones is of high importance, as failures can have critical consequences. One of the key strategies in safety assurance is to capture the system's properties describing what the system should and should not do under different conditions. Encoding the requirements as formal or semi-formal properties, can enable creating safety and security guardrails for system behavior. Formulating properties usually starts from the system requirements typically written in natural language. Therefore, large language models (LLMs) with their strong potential can be leveraged to extract properties from existing documentation and software code. These properties can be used to drive subsequent automated testing and verification activities, such as property-based testing [4,6]. Property-based tests (PBT) are generated from a given property describing the expected behavior, and the testing framework produces various input scenarios to check if the property is satisfied in all cases.

In this paper, we propose a novel automated and scalable approach for guardrailing CPS using PBT generated by LLMs. Our approach benefits from two major established facts in software engineering: 1) the cyber side of CPSs is essentially a software program amenable to existing automated program analysis tools; and 2) advanced LLMs are strong in analyzing programs and extracting their expected properties [22]. Based on these two observation, our proposed approach, called ChekProp, uses LLMs to generate property-based tests for CPSs before their deployment, i.e., at design time. These PBTs can also then be utilized after deployment, i.e., at run time, to detect unsafe behavior of the CPS. ChekProp uses the source code, documentation, and unit tests of the target CPS to extract *properties* regarding its expected behavior. ChekProp also generates PBTs that verify that the extracted properties hold for the CPS. We implement a prototype of ChekProp and make it publicly available to the community [5].

Our work is the first to address property-based testing of CPS. In software systems context, LLMs have been utilized for automated test case generation, both unit and integration test, from various sources of specification. For CPS, efforts have been more focused on scenario generation for autonomous driving and robotics (not inferring formal properties or invariants for the system). Using LLMs for generating property-based testing has recently emerged like in the work of Vikram et al. (2024) which investigates if LLMs write good PBTs [22]. In this work, the system under test is a Python library. CPSs are supposed to run in a physical environment. This makes the generation of PBT more challenging than just extracting the property or testing software. For generating an executable PBT, we need a detailed understanding of the relationship between various components of a given CPS to be able to mock how the environment affects the CPS program. A different approach for using LLMs to ensure the safety of CPS is to express the requirements of the system in the form of mathematical equations between different physical components. For example, Abshari et al. [1] propose an LLM-based approach that automatically extracts physical invariants that ensure healthy execution of the system. This technique is focused

Listing 1. An example of a Python property-based test that checks the `pow` method returns a positive number as the square of an integer. This PBT uses the `hypothesis` library to generate inputs for the test.

```
The property-based unit test for the pow method:
@given(st.integers())
def test_pow2_positive_output(x):
    square = pow(x, 2)
    assert square >= 0

The example-based unit test for the pow method:
def test_pow2_on_negative_input()
    square = pow(-3, 2)
    assert square == 9
```

only on extracting the invariants (properties), while CHEKPROP generates fully executable property-based tests for CPS.

We evaluate the relevance of properties extracted by CHEKPROP for nine programs: two widely studied CPSs and seven Raspberry Pi programs. CHEKPROP extracts 25 properties on these nine programs. Our results show that the properties extracted by CHEKPROP are similar to those carefully created with manual effort, with a recall of 94% and a precision of 72%. The high precision and recall of CHEKPROP shows that it can be a reliable tool for automating the manual effort dedicated to property extraction for CPSs. Moreover, we study the quality of PBTs generated by CHEKPROP. We find that 47% of the PBTs generated by CHEKPROP become executable with minor modifications and 85% of the PBTs effectively cover various parts of the input space partitions. This suggests that CHEKPROP generates PBTs that successfully verify the CPS compliance with the extracted properties.

In summary, our main contributions are the following.

- We propose a novel automated approach for generating property-based tests for CPSs using LLMs.
- We implement a prototype of our proposed approach in CHEKPROP and make it accessible to the community in our open-source repository [5].
- We report the results of our preliminary experiments on the relevance and quality of the PBTs generated by CHEKPROP in practice.

2 Background on Property-Based Testing

Property-based testing was first introduced in QuickCheck [4]. Given a function under test f, the input space of this function X, and a property P that checks the behavior of f on a given input, a property-based test validates that $\forall x \in X : P(x, f)$. The property P can be seen as a function that takes an input x and the function f and outputs `true` or `false`. The output of P determines if f behaves according to predefined requirements on x. In practice, a property-based test (PBT) consists of three components: 1) an *input generator* `gen()`, which returns different inputs, like x, from the input space X; 2) a *test body* that collects relevant data regarding the behavior of f; and 3) a *test assertion* that uses the data collected by the test body to assert that the property P about f is `true` for a given input x.

Take Listing 1 as an example. The PBT in this example (lines #1–5) tests the Python `pow` method. The main goal of this test is to check that the `pow` method returns a positive number when it powers an integer by 2. The PBT employs the `hypothesis` library [12] for input generation (line #2). The `given` decorator at line #2 uses the `st.integers()` strategy to generate various random integers and consider them as input x at line #3. The `hypothesis` library provides the `given` decorator, the `st.integers()` strategy, and many other tools to facilitate property-based testing in Python. After the input is generated with the help of `hypothesis` library, the PBT in Listing 1 collects the output of `pow` and saves it in `square` at line #4. Finally, it checks the property that the output of `pow` is positive at line #5. This property-based test delivers exactly what we need: a test that checks that the pow method returns a positive number for various integers, when they are powered by 2.

To better understand property-based testing, we can compare it with the commonly used example-based unit tests (EBTs), which test program with fixed arguments [20]. An EBT checks if the function f under test works correctly for a single input x. For this, the EBT usually inspects that the output of f for x is exactly the same as the correct output o determined by an oracle. In contrast, a PBT checks that f behaves according to expectations for various inputs from the input space X.

Listing 1 shows an example of EBT for the Python `pow` method as well. The main goal of this test is also checking that the `pow` method returns a positive number when it powers an integer by 2. The EBT (lines #7–10) tests the `pow` method by giving it a negative integer, namely -3, as base, and 2 as power. Then, it checks that the output of `pow` is exactly 9 (line #10). If this test passes, it only shows that `pow` method returns a positive number as the square of -3 (as an example of an integer). The EBT is testing `pow` only for one single input, and its result might not be generalizable. Also, this type of testing requires an oracle that states the expected output, which is 9 in this case.

PBTs are suitable for safety checking the behavior of cyber-physical systems at runtime for two reasons. First, in PBTs, we do not need to know the exact expected behavior of the CPS under test. PBTs only check that the behavior of the CPS meets certain requirements, which can indicate the safety of its behavior. Secondly, PBTs check the behavior of the CPS on any input from the input space. This enables PBTs to ensure the safety of the CPS under unforeseen situations at runtime. We need to utilize the test body and test assertion of the PBT to monitor and assure the safety of CPS behavior at runtime. Based on these observations, in this paper, we use LLMs to generate PBTs for cyber-physical systems.

3 Proposed Approach

We envisage a novel approach for guardrailing cyber-physical systems with LLM-generated PBTs. Figure 1 illustrates an overview of our proposed approach. This approach consists of two main phases: the PBT generation phase, and the property-based monitoring phase.

The PBT generation phase occurs at *design time, before the system is deployed in real world*. In this phase, we employ LLMs to generate PBTs that guardrail cyber-physical systems against running in unwanted/unsafe states. We implement our tool CHEKPROP to carry out the proposed PBT generation, given the documents, code, and unit tests of the cyber-physical system. After the PBTs are generated, we enter the property-based monitoring phase of our approach. This phase occurs at *run time, when the system is deployed in real world*. In this phase, the running cyber-physical system is constantly checked against the generated PBTs. Once a violation of a PBT is detected, a warning is raised and the proper safety measures should be taken.

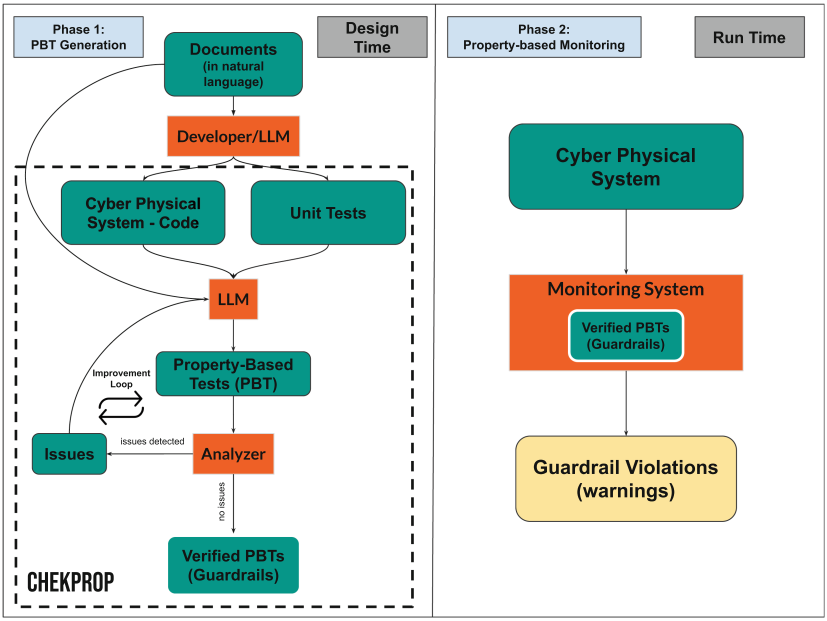


Fig. 1. Overview of the proposed two-phase approach. CHEKPROP particularly focuses on the PBT generation phase. In the PBT improvement loop step, CHEKPROP aims to improve the generated PBTs by sending the LLM an *improvement prompt*, consisting of the failed PBTs and the error messages collected for them. Errors of various types are considered, including syntax errors, compilation errors, exceptions thrown during test executions, and assertion failures.

In the following, we explain the proposed PBT generation method, which is implemented in ChekProp. ChekProp takes the natural language documents of the CPS, its source code, and unit tests and prompts an LLM to generate PBTs. Next, it analyzes existing PBTs to detect issues and iteratively prompts the LLM to improve the PBTs. Once the PBTs pass the analysis, they are considered as verified PBTs that can be used as guardrails for the CPS. We now discuss each of these ChekProp components in more detail.

3.1 Inputs of *ChekProp*

The input to ChekProp comprises natural language documents that specify the CPS and its expected behavior, the CPS source code in Python, and unit tests for this source code.

The natural language documents of the CPS describe the expected behavior of the system and the constraints that should be observed in the run time. Take Fig. 2 as an example of a natural language document that describes a Pneumatic Control System (PCS) [13]. The description first defines the main elements involved in the system, namely, the horizontal and vertical cylinders and their corresponding sensors and controllers. Next, it explains the expected behavior of the system and the expected order of cylinder movements. Finally, it presents the constraints that should be met during the movements.

Given the natural language documents of the system, the CPS is implemented to follow the described requirements. Moreover, a set of unit tests is created to test the CPS implement for specific points in the input space. Note that implementing the CPS and creating unit tests for it can also be fully automated using state-of-the-art LLM-based code generation [9] and test generation [23] techniques. However, ChekProp focuses on PBT generation and assumes that the CPS implementation is provided in Python, along with at least one unit test that demonstrates how the test body should interact with different methods of the program.

This module contains the implementation of a Pneumatic Control System (PCS) system. The PCS system consists of two cylinders, each with a sensor attached to it. The system has two controllers, one for each cylinder. One of the cylinders is vertical and the other is horizontal.

The vertical cylinder can move up and down, while the horizontal cylinder can move left and right. Each cylinder should move between locations 0 and 2. Controller A is responsible for controlling the horizontal cylinder, while Controller B is responsible for controlling the vertical cylinder. Location 0 is the starting location for both cylinders, which is top-left in the system. The movement should follow this pattern: vertical cylinder moves down, vertical cylinder moves up, horizontal cylinder moves right, vertical cylinder moves down, vertical cylinder moves up, horizontal cylinder moves left.

When the vertical cylinder is at bottom, the horizontal cylinder should not move. Also, going beyond the 0 to 2 location interval will break the cylinder.

Fig. 2. The natural language document that describes a Pneumatic Control System (PCS). We take the original design of PCS from [13] and adapt it to make it suitable for property-based testing.

3.2 Initial PBT Generation

ChekProp starts PBT generation by synthesizing an *initial prompt*. This prompt is used to invoke the LLM for generating an initial batch of PBTs. The initial prompt consists of four main sections and follows the structure presented in Fig. 3.

As illustrated in the Fig. 3 example, the first section presents the description of the CPS in natural language. LLMs are highly effective in understanding natural language specifications of software and translating those specifications to actual code [9]. Therefore, we provide this section of the prompt to help the LLM better understand the system constraints that should be later translated into PBTs.

The second part of the initial prompt contains the Python code for the CPS. In Fig. 3, the second section presents a part of the code for pneumatic control system, namely, the `Cylinder` class (line #14). Including the system code in the prompt is essential for the LLM to recognize how the system should be called in tests.

The third part of the initial prompt also provides at least one example unit test for the system. The third part of Fig. 3 shows an example of a unit test that calls the system controller. This unit test employs an instance of the `MockSystem` class (line #58) to mock the physical part of the pneumatic control system and obtain a simple interface to its controller. It also illustrates how the states of the system should be collected during execution and checked later (lines #59–61).

Note that, as explained in Sect. 2, there is significant difference between unit tests and property based tests. The unit test only checks the behavior of the program for a specific input. For example, in the unit test (Sect. 3) of Fig. 3, specific `total_time`, `cylinder_interval`, etc. are used. Also, in unit testing we usually check that the output is exactly what is expected according to an oracle [21]. In contrast, property based tests check that a more general condition is meet by the program behavior over a wide range of inputs.

The fourth and final part of the Fig. 3 instructs the LLM to generate the desired PBTs. At the end of the initial PBT generation step, ChekProp obtains a set of initial PBTs. ChekProp runs these PBTs and collects their results using an analyzer unit. If a group of generated PBTs fails, ChekProp collects their failure message and enters its PBT improvement loop step as described in Subsect. 3.3.

3.3 PBT Improvement Loop

In the PBT improvement loop step, ChekProp aims to improve the suite of generated PBTs. For this, ChekProp sends the LLM an *improvement prompt*, consisting of the failed PBTs and the error messages collected for them. Errors of various types are considered, including syntax errors, compilation errors, exceptions thrown during test executions, and assertion failures. The improvement prompts are sent to the LLM in continuation of the initial prompt, which means

Section 1: Natural Language Description

```
The following is a description of a pneumatic control system.

{Natural Language Description}
---------------------------------------------------------------------------
The following code implements this pneumatic control system. You should generate property based tests for this code.

```python
import math
import threading
from time import sleep

....

class Cylinder:
    def __init__(self, sensor: Sensor):
        self.motion = 0
        self.sensor = sensor
        self.just_stopped = False

    def trigger_motion(self):
        if self.is_at_start():
            self.motion = 1
        elif self.is_at_end():
            self.motion = -1

    def move(self):
        self.sensor.location = self.sensor.location + self.motion
        self.just_stopped = False

    def start_working(self, total_time: float, cylinder_interval: float):
        for i in range(math.floor(total_time / cylinder_interval)):
            if self.motion != 0:
                self.move()
                if self.is_on_border():
                    self.motion = 0 # Stop movement
                    self.just_stopped = True
            sleep(cylinder_interval)

    def is_on_border(self):
        return self.is_at_end() or self.is_at_start()

    def is_at_end(self):
        return self.sensor.location == 2

    def is_at_start(self):
        return self.sensor.location == 0

    ....

```

The following is a unit test for this system class:

```python
from examples.gpiozero.apps.pcs.src.pcs import MockSystem

def test_starting_motion():
    mock_system = MockSystem(total_time=1, cylinder_interval=1,
controller_interval=1, mock_interval=1)
    collected_states = mock_system.execute_scenario()
    assert collected_states[0].cylinder_a_motion == 0
    assert collected_states[0].cylinder_b_motion == 1
```

Generate property based tests for this system following the steps below:
    1. Based on the given description and code, extract the properties of the system.
    2. Use the unit tests to understand the behavior and interface of the code.
    3. Based on the extracted properties and your understanding of the code, use the hypothesis
       library to generate property based tests.
```

Section 2: Cyber Physical System Code

Section 3: Unit Test for the CPS Code

Section 4: Instruction for PBT Generation

Fig. 3. The structure of initial prompt used by CHEKPROP for generating an initial set of PBTs.

Listing 2. The PBT generated by CHEKPROP for the Pneumatic Control System presented in Figure 3 with three parts, the input generator, the test body, and the test assertions.

```
1 from hypothesis import given, strategies as st
2 from examples.gpiozero.apps.pcs.src.pcs import MockSystem
3
4
5 # Test that the cylinder locations remain within the bounds (0 and 2) at all times.
6 @given( # This is the input generator of the PBT, which uses the hypothesis library to run the test with different inputs.
7     total_time=st.floats(min_value=1.0, max_value=100.0),
8     cylinder_interval=st.floats(min_value=0.1, max_value=10.0),
9     controller_interval=st.floats(min_value=0.1, max_value=10.0),
10    mock_interval=st.floats(min_value=0.1, max_value=10.0),
11 )
12 def test_cylinder_location_in_bounds(total_time, cylinder_interval, controller_interval, mock_interval):
13     system = MockSystem(total_time, cylinder_interval, controller_interval, mock_interval) # Start of test body
14     collected_states = system.execute_scenario()
15
16     for state in collected_states: # End of test body
17         # The following two lines present the test assertions of this PBT
18         assert 0 <= state.cylinder_a_loc <= 2, f"Cylinder A out of bounds: {state.cylinder_a_loc}"
19         assert 0 <= state.cylinder_b_location <= 2, f"Cylinder B out of bounds: {state.cylinder_b_location}"
```

that the LLM also has the CPS description, code, and unit test in context. Therefore, the LLM has all the information needed to improve the PBTs.

The PBT improvement loop component of CHEKPROP, iteratively sends improvement prompts to the LLM and employs the analyzer unit to run the improved PBTs and collect their results. If the improved PBTs still fail, CHEKPROP repeats this process until all PBTs are fixed or CHEKPROP reaches a predefined maximum number of improvement attempts.

3.4 Output of *CHEKPROP*

In the event of a successful PBT generation, CHEKPROP outputs the PBT as Python code. In particular, per our experiments, LLMs always use the `hypothesis` library [12] to write property based tests in Python. The tests generally use the same testing framework as the example unit test. For example, the unit test in Fig. 3 is run with `pytest`. The PBTs generated by LLMs for this prompt can also be executed with `pytest`.

Listing 2 shows one of the PBTs that CHEKPROP generates for the pneumatic control system. As the comment above the test mentions, it checks that cylinders stay within the location bound. The property is checked by two assertions at lines #17–18, one assertion per cylinder. This property is checked for a range of different settings for cylinders and their controllers presented at lines #7–10. Note the main difference between this PBT and the unit test in Fig. 3: the unit test checks that a specific output for a given input is exactly correct, while the PBT verifies that a general property holds for all inputs within a specified range. This makes PBTs more general and appropriate for checking that the system does not show unsafe behavior in unforeseen situations.

3.5 Implementation

CHEKPROP uses the `gemini-2.0-flash-lite-preview-02-05` in its current version, but adopts a flexible design that allows easy switch to other LLMs. CHEKPROP also invokes the LLM with a sample size of one and a temperature of zero, which means that it receives only the top response per LLM invocation. In the current version of CHEKPROP, the PBT improvement loop is disabled, and we assess LLM's ability to generate PBTs at the initial attempt.

3.6 Property-Based Monitoring

While CHEKPROP is focused on PBT generation (phase 1 in Fig. 1), property-based monitoring (phase 2 in Fig. 1) is also an integral part of our guardrailing CPSs with PBTs. In the monitoring phase of our proposed approach, various components of the generated PBTs are used to collect relevant data and verify the properties at runtime. For example, the generated PBT in Listing 2 tests that the cylinders stay in the correct location range for given inputs. This test shows that we should collect `state` values (line #16) and then assert that their `cylinder_a_loc` attribute is in the range $[0, 2]$ (line #17) to ensure the property holds for cylinder A. In the monitoring phase, we can use the same data collection and property assertion techniques to check that the cylinders do not enter unsafe locations at runtime. This example demonstrates how the property extracted by CHEKPROP and the implementation of its generated PBTs are useful, relevant, and vital for guardrailing CPSs.

More generally, as explained in Sect. 2 and as seen in Listing 2, the PBTs generated in the first phase consist of three components: an input generator, a test body, and a test assertion. In the property-based monitoring phase, the PBT input generator is no longer needed, as the inputs are generated by the cyber part (in the form of control commands) and the physical system (via sensor values). The test body is replaced by the monitor that sits between the controller and the physical system collecting relevant data. The properties derived at design time are transformed into guards that are checked at runtime. The monitor verifies that these guards hold in the current state of the system. Based on the situation, if a violation is detected, the monitor can intercept and block the command being sent to the physical system. In this way, the PBTs generated by CHEKPROP serve as runtime guardrails for CPSs.

4 Experiments

4.1 Research Questions

We conduct preliminary experiments to answer the following research questions:

In this paper, we study the quality of CHEKPROP extracted properties and the quality of its generated property-based tests according to the following research questions:

- **RQ1** (Property relevance): Does the proposed approach extract relevant properties? We assess the quality of properties extracted by our approach on a dataset of Python programs for cyber-physical systems. Our dataset consists of two cyber-physical systems that are extensively studied in the literature, as well as seven Raspberry Pi programs. We compare the automatically extracted properties with manually crafted properties to judge their relevance.
- **RQ2** (PBT quality): What is the quality of CHEKPROP generated PBTs for real-world CPSs? We use CHEKPROP to generate PBTs for Python CPS programs in our dataset. We assess the quality of CHEKPROP generated PBTs from two aspects. First, we check if the generated PBTs can be executed with minimal manual modification (executability). Second, we evaluate the extent of various input space partitions on which the generated PBTs execute the program (effectiveness in terms of coverage of the input space partitions).

4.2 Dataset

As the current version of CHEKPROP supports PBT generation for Python programs, we curate a dataset of Python CPS programs for our experiments. This dataset consists of nine programs presented in Table 1. These programs are taken from three main sources as follows.

First, we include the Python version of two CPSs that are widely studied in the model checking literature [13]: a temperature control system (TCS) and a pneumatic control system (PCS) (P1 and P2 in Table 1). TCS and PCS are presented with the IDs P1 and P2 in Table 1. Moradi et al. [13] use the model checking tool of Rebeca[1], Afra [17], to detect potential attacks against these systems. For this, they define the correctness properties for each system and assess if Afra can find counterexamples for these properties on models augmented with malicious behavior. The manually defined properties in [13] set a ground-truth with which we can compare the properties automatically extracted by CHEKPROP.

We carefully implement TCS and PCS in Python to make them amenable to PBT generation by CHEKPROP. Listing 3 shows a summary of our implementation of the TCS. There is a class for each component (i.e. Rebeca actor) of the system, namely, `TempSensor` for the sensor, `HCUnit` for the HC unit, and `Controller` for the controller. Each component runs on a separate thread and updates its status periodically. For example, the sensor fetches current temperature every `sensor_interval` seconds (see line #17). We also provide a `MockRoom` class that simulates the room environment and enables us to execute the system with different configurations, such as `initial_temp` and `sensor_interval` (see lines #50 and #58). This implementation is suitable for testing the temperature control system.

[1] Rebeca is an actor-based modeling language, in which actors are the units of concurrency and communicate via asynchronous message passing [17].

Table 1. The cyber-physical system Python programs considered in our dataset.

ID	Program	Type	Description
P1	Temperature Control System (TCS)	Model-based Example [13]	The TCS system controls the temperature of a room. The room has a window that blows cold or hot wind into the room. A sensor collects the current temperature and a HC unit is used to heat up or cool down the room environment. Finally, a controller collects the temperature data from the sensor and decides if the HC unit should be used.
P2	Pneumatic Control System (PCS)	Model-based Example [13]	The PCS regulates the movement of components of a mechanical system that consist of a horizontal and a vertical cylinder, their corresponding controllers and sensors. This mechanical system is designed to pick an object from the ground and move it to a new place.
P3	Laser Tripwire	Raspberry Pi Projects [16]	A prototype on a breadboard to detect whether a beam of light is hitting the light-dependent resistor (LDR).
P4	Line Following Robot	Raspberry Pi Projects [16]	A robot with two motors and two sensors. The robot uses its right and left sensors to follow a line and decide if should go turn right, left, or go forward.
P5	Ultrasonic Theremin	Raspberry Pi Projects [16]	A system that produces a sound with a volume level corresponding to the user distance from the device.
P6	Remote Buggy	Raspberry Pi Projects [16]	A robot that moves in four directions and a user remotely controls it.
P7	Quick Reaction Game	Raspberry Pi Projects [16]	A game in which a light turns on and off and two players compete to hit their button faster after the light is turned off.
P8	Presence Indicator	Raspberry Pi Projects [16]	A monitor made of a list of LEDs that shows the number of people present at a place.
P9	GPIOZERO InputDevice	GPIOZERO Library Class [19]	A core class in GPIOZERO library that handles connection to input devices on a Raspberry Pi board.

The second set of programs in our dataset are the six programs (P3-P8 in Table 1) taken from open-source Raspberry Pi projects [16] that use the GPIOZERO library [19]. GPIOZERO is a Python library that real-world cyber-physical systems employ to connect to Raspberry Pi boards. We take the source of these programs from the official Raspberry Pi website [16] and manually add unit tests for them. These unit tests can be used in CHEKPROP prompts (see Subsect. 3.1). The six programs in our dataset that use GPIOZERO enable us to evaluate CHEKPROP applicability on projects that adopt the widely used Raspberry Pi boards.

The last program considered in our dataset (P9 in Table 1) is `InputDevice`, a core class from the GPIOZERO library. This class provides an interface for Python programs to interact with Raspberry Pi input devices, such as barometers, temperature sensors, etc. The `InputDevice` class is a complex class from GPIOZERO that connects to various components in this library. Testing this class requires a detailed understanding of the inner workings of the library and how it represents and handles the physical environment. For example, to instantiate an object of the `InputDevice` class, we have to pass the number of a *pin* to its constructor method. The type of this pin should be "input", while some of the pins on a Raspberry Pi board are reserved only for "output". A correct test should use a pin of the correct type to instantiate an `InputDevice`. Another example of such details in GPIOZERO is how pins are activated. To activate a pin on a Raspberry Pi board, its voltage should go high, which can happen by calling the `pin.drive_high()` method. Writing a test for a CPS program requires an accurate understanding of how interacting with the program, such as calling `pin.drive_high()`, impacts the physical status of the system, such as increasing the voltage on the Raspberry Pi board. To evaluate whether CHEKPROP generated PBTs capture such details about CPS programs, we include the `InputDevice` class in our dataset.

Listing 4 presents the `InputDevice` class. This class, similar to other GPIOZERO classes, has well-written documentation (lines #2–39). We consider this documentation as the natural language description in CHEKPROP prompts (see Subsect. 3.1). Moreover, GPIOZERO has extensive unit tests for its classes, which we use them to produce our initial prompt . This confirms that GPIOZERO classes have the essential components for applying CHEKPROP: the natural language description, the source code, and the unit test (see Subsect. 3.1).

Overall, our dataset contains a combination of CPSs studied in research literature and CPS programs that employ widely used libraries. This dataset helps us to assess the relevance of CHEKPROP extracted properties and the quality of its generated PBTs.

4.3 RQ1: Property Relevance

I. Methodology: To assess the quality of extracted properties, we run our approach on the nine programs in our dataset and analyze the relevance of the extracted properties. In this experiment, we abstract away the implementation details of the generated PBTs; instead, we focus on how well the properties considered in the PBTs validate the logic of the program under test. For this purpose, we compare properties extracted by our approach with manually crafted properties that we consider as ground-truth. In particular, we evaluate whether the logic checked by ground-truth properties is also validated by CHEKPROP extracted properties and vice versa. As explained in Subsect. 4.2, for programs P1 and P2, the ground-truth properties are already stated by Moradi et al. [13]. For the remaining programs (P3–P9), we manually define the ground-truth properties.

Listing 3. Implementation of the temperature control system in Python.

```python
class Environment:
    def __init__(self, initial_temp: int = None):
        self.temp = initial_temp if initial_temp is not None else random.randint(20, 24)

    def fetch_temp(self):
        return self.temp
    ....

class TempSensor:
    def __init__(self, env: Environment):
        self.env = env
        self.temp = self.env.fetch_temp()

    def start_temp_collection(self, total_time: float, sensor_interval: float):
        for i in range(math.floor(total_time / sensor_interval)):
            self.temp = self.env.fetch_temp()
            sleep(sensor_interval)

class PWMOutputDevice:
    .....

class HCUnit:
    def __init__(self):
        self.cooler = PWMOutputDevice()
        self.heater = PWMOutputDevice()

    def activate_cooler(self):
        self.cooler.on()
        self.heater.off()
    ....

class Controller:
    def __init__(self, temp_sensor: TempSensor, hc_unit: HCUnit):
        self.temp_sensor = temp_sensor
        self.hc_unit = hc_unit

    def control(self, total_time: float, control_interval: float):
        for i in range(math.floor(total_time / control_interval)):
            temperature = self.temp_sensor.temp
            if 21 <= temperature <= 23:
                self.hc_unit.deactivate()
    ...

class SystemState:
    def __init__(self, temp, cooler_state, heater_state, outside_air_temp):
        self.temp = temp
    ...

class MockRoom:
    def __init__(self, total_time: float, sensor_interval: float, control_interval: float,
      initial_temp: int = None):
        self.env = Environment(initial_temp=initial_temp)
        self.total_time = total_time
        self.sensor_interval = sensor_interval
        self.control_interval = control_interval
        self.temp_sensor = TempSensor(self.env)
        ...

    def execute_scenario(self):

        sensor_thread = threading.Thread(target=self.temp_sensor.start_temp_collection,
                                         args=(self.total_time, self.sensor_interval))
        sensor_thread.start()
        ...

        collected_states = []
        for i in range(self.total_time):
            ...
            outside_air_temp = self.env.get_outside_air_temp()
            collected_states.append(SystemState(cur_temp, ...))
            self.env.set_temp(cur_temp + outside_air_temp + heater_value - cooler_value)
            sleep(1)

        sensor_thread.join()
        control_thread.join()

        return collected_states
```

Listing 4. The `InputDevice` class in GPIOZERO.

```python
class InputDevice(GPIODevice):
    """
    Represents a generic GPIO input device.

    This class extends :class:`GPIODevice` to add facilities common to GPIO
    input devices.  The constructor adds the optional *pull_up* parameter to
    specify how the pin should be pulled by the internal resistors. The
    :attr:`is_active` property is adjusted accordingly so that :data:`True`
    still means active regardless of the *pull_up* setting.

    :type pin: int or str
    :param pin:
        The GPIO pin that the device is connected to. See :ref:`pin-numbering`
        for valid pin numbers. If this is :data:`None` a :exc:`GPIODeviceError`
        will be raised.

    :type pull_up: bool or None
    :param pull_up:
        If :data:`True`, the pin will be pulled high with an internal resistor.
        If :data:`False` (the default), the pin will be pulled low.  If
        :data:`None`, the pin will be floating. As gpiozero cannot
        automatically guess the active state when not pulling the pin, the
        *active_state* parameter must be passed.

    :type active_state: bool or None
    :param active_state:
        If :data:`True`, when the hardware pin state is ``HIGH``, the software
        pin is ``HIGH``. If :data:`False`, the input polarity is reversed: when
        the hardware pin state is ``HIGH``, the software pin state is ``LOW``.
        Use this parameter to set the active state of the underlying pin when
        configuring it as not pulled (when *pull_up* is :data:`None`). When
        *pull_up* is :data:`True` or :data:`False`, the active state is
        automatically set to the proper value.

    :type pin_factory: Factory or None
    :param pin_factory:
        See :doc:`api_pins` for more information (this is an advanced feature
        which most users can ignore).
    """
    def __init__(self, pin=None, *, pull_up=False, active_state=None,
                 pin_factory=None):
        super().__init__(pin, pin_factory=pin_factory)
        try:
            self.pin.function = 'input'
            pull = {None: 'floating', True: 'up', False: 'down'}[pull_up]
            if self.pin.pull != pull:
                self.pin.pull = pull
        except:
            self.close()
            raise

        if pull_up is None:
            if active_state is None:
                raise PinInvalidState(
                    f'Pin {self.pin.info.name} is defined as floating, but '
                    f'"active_state" is not defined')
            self._active_state = bool(active_state)
        else:
            if active_state is not None:
                raise PinInvalidState(
                    f'Pin {self.pin.info.name} is not floating, but '
                    f'"active_state" is not None')
            self._active_state = False if pull_up else True
        self._inactive_state = not self._active_state

    @property
    def pull_up(self):
        """
        If :data:`True`, the device uses a pull-up resistor to set the GPIO pin
        "high" by default.
        """
        pull = self.pin.pull
        if pull == 'floating':
            return None
        else:
            return pull == 'up'
    ...
```

Take the temperature control system (TCS) as an example. Moradi et al. outline three properties for TCS as follows:

1. If the room is warm (temp $>$ 23), the HC unit should not be heating the room.
2. If the room is cold (temp $<$ 21), the HC unit should not be cooling the room.
3. The temperature should never be too low (temp $<$ 20) or too high (temp $>$ 24).

We first apply our proposed approach on our Python implementation of TCS to generate PBTs. Next, we compare the extracted properties that are tested in these PBTs with the three ground-truth properties in [13]. If the CHEKPROP properties correspond with the three ground-truth properties, we conclude that the proposed approach is able to extract useful properties.

II. Results: Table 2 shows the results of this experiment. In total, the table contains 26 properties. We split these properties into four groups: Group1 consists of 15 properties that are present among ground-truth and CHEKPROP extracted properties in the exact same form (Pr3, Pr5, Pr7, Pr9, Pr10, Pr11, Pr12, Pr14, Pr15, Pr17, Pr20, Pr21, Pr23, Pr24, and Pr25); Group2 consists of 3 properties that are present among ground-truth and CHEKPROP extracted properties in equivalent but slightly different forms (Pr1, Pr2, and Pr6); Group3 consists of 7 properties that are only among the CHEKPROP extracted properties (Pr4, Pr8, Pr13, Pr16, Pr19, Pr22, and Pr26); and Group4 consists of 1 property that is only among ground-truth properties (Pr18).

In total, the ground-truth contains 19 properties (Group1+Group2+Group4) and CHEKPROP extracts 25 properties (Group1+Group2+Group3). Among all properties, 18 are common between ground-truth and CHEKPROP, either in the exact same form (Group1) or with *distinct* different formulations (Group2). These properties are relevant, since they are present among the manually crafted properties. Therefore, the recall of CHEKPROP is 94% (18/19), which indicates that our approach can fully replace the manual effort required for extracting most of the properties from CPSs. The precision of CHEKPROP is 72% (18/25), suggesting that the properties extracted by CHEKPROP often represent what humans expect from the CPS under test. The high precision and recall of CHEKPROP make it a reliable tool for automating the manual effort dedicated to property extraction for CPSs.

In Table 2, we see that in three cases (Pr4, Pr8 and Pr26) CHEKPROP extracts a property that is relevant and useful, but neglected in manually crafted properties. For example, our approach extracts Pr4 for PCS which notes that neither heater or cooler should be active when the room temperature is between 21 °C and 23 °C. This indicates that not only can our automated approach replace manual property design work, but it can also even improve the manually crafted properties.

The three relevant properties neglected in ground-truth together with the 18 properties common between ground-truth and CHEKPROP make up the set of our 21 relevant properties.

Table 2. Comparison between properties automatically extracted by our proposed approach and ground-truth properties that are manually crafted.

ID	Program	Ground-truth Property	Corresponding CHEKPROP Property
Pr1	TCS	If the room is warm (temp > 23), the HC unit should not be heating the room.	Heater should be activated only if the temperature drops below 21°C.
Pr2	TCS	If the room is cold (temp < 21), the HC unit should not be cooling the room.	Cooler should be activated only if the temperature exceeds 23°C.
Pr3	TCS	The temperature should never be too low (temp < 20) or too high (temp > 24).	Same as ground-truth.
Pr4	TCS	Neglected.	Neither heater nor cooler is active when temperature is in the target range (21-23°C).
Pr5	PCS	The horizontal cylinder should not move when the vertical cylinder is down.	Same as ground-truth.
Pr6	PCS	The horizontal and vertical cylinders should not move simultaneously.	The movement should follow a specific order.
Pr7	PCS	The cylinders location should always be between 0 and 2.	Same as ground-truth.
Pr8	PCS	Neglected.	The cylinder movement speed should not exceed 1.
Pr9	Laser Tripwire	"INTRUDER" is printed if an only if there is no light.	Same as ground-truth.
Pr10	Line Following Robot	When left sensor is on, left motor goes backwards and right motor goes forward.	Same as ground-truth.
Pr11	Line Following Robot	When right sensor is on, right motor goes backwards and left motor goes forward.	Same as ground-truth.
Pr12	Line Following Robot	When both sensors are off, both motors go forward.	Same as ground-truth.
Pr13	Line Following Robot	Correctly not included.	When both sensors are on, at least one motor is moving. *This is an irrelevant property that never occurs in practice.*
Pr14	UltrasonicThermin	Volume level never gets too high or too low.	Same as ground-truth.
Pr15	UltrasonicThermin	Volume increases when user is getting closer to the sensor.	Same as ground-truth.
Pr16	UltrasonicThermin	Correctly not included.	The buzzer has a correct range of volume. *This is an irrelevant property that mostly checks the inner classes of the GPIOZERO library, not the application.*
Pr17	Remote Buggy	Pressing each button on the controller, activates the corresponding motor.	Same as ground-truth.
Pr18	Quick Reaction Game	Check that the light turns on at some point and turns off afterwards.	Not extracted.
Pr19	Quick Reaction Game	Correctly not included.	When the `valid_pressed` method is called, the name of the winner is printed. *This is not a useful property, as it only tests very detailed implementation details.*
Pr20	PresenceIndicator	The LEDs show the number of present people divided by 10.	Same as ground-truth.
Pr21	PresenceIndicator	When the number of present people is above 10, the number "1" is shown.	Same as ground-truth.
Pr22	PresenceIndicator	Correctly not included.	When the `PresneceIndicator` object is closed, the LEDs are off. *This is not a useful property, as it only tests very detailed implementation details.*
Pr23	GPIOZEROInputDevice	The `pull_up` parameter of the input device affects the pull state of the pin.	Same as ground-truth.
Pr24	GPIOZEROInputDevice	With a `none` `pull_up`, the `active_state` should be set.	Same as ground-truth.
Pr25	GPIOZEROInputDevice	When `pull_up` is set, the `is_active` parameter has a reverse effect between the `pull_up` value of the device and the activation of the pin.	Same as ground-truth.
Pr26	GPIOZEROInputDevice	Neglected.	When the input device is closed, the pin is not in use anymore.

As presented in Table 2, four of the properties extracted by CHEKPROP (Pr13, Pr16, Pr19, and Pr22) are not useful. These properties either check a state that does not occur in real-world (Pr13) or validate highly detailed implementation nuances. This observation shows that a manual check on properties automatically extracted by CHEKPROP is needed to ensure that no useful property is considered for testing.

Finally, there is only one ground-truth property (Pr18) that does not correspond to any of the properties extracted by CHEKPROP. Pr18 is a property for the quick reaction game and indicates the order of changes in the light status, it should be first turned on at some point and then turned off at some point. With a careful manual analysis, we understand that the code we provide to the LLM for the quick reaction game lacks the documentation regarding this point. This suggests the importance natural language description as of one core components in CHEKPROP prompts (see Subsect. 3.1).

Answer to RQ1: Does the proposed approach extract relevant properties?
We compare the manually crafted relevant properties with CHEKPROP extracted properties for nine programs in our dataset. This comparison shows that 94% (18/19) of the ground-truth properties are also automatically extracted by CHEKPROP. Moreover, CHEKPROP extracts three additional relevant properties that are neglected in manually crafted properties. This indicates that CHEKPROP is a reliable tool for automating the tedious and complicated task of defining CPS properties..

4.4 RQ2: PBT Quality

I. Methodology: For evaluating the quality of PBTs generated by CHEKPROP, we examine the PBTs that test the 21 relevant properties according to our analysis in the RQ1 experiment (see Subsect. 4.3). As explained in Subsect. 4.1, we assess the applicability of our approach from two aspects: executability and effectiveness.

We consider a PBT executable if and only if it is correct both syntactically (i.e., successfully compiles) and semantically (i.e. passes). To investigate the executability of a PBT, we check to what extent the PBT should be manually modified to reach syntax and semantic correctness. A lower level of manual modification indicates higher executability and vice versa. We perform a manual analysis to find the level of executability of PBTs. Based on this analysis, we assign the PBTs generated for each program to one of the following executability levels: “HIGH”, “MED”, and “LOW”. A “HIGH” executability level means that the analyzer has to spend less than one minute manually fixing the PBT to ensure it runs and passes successfully. “LOW” means that more than three minutes of manual work is needed, and “MED” means that between one and three minutes is required. In this investigation, for the PBTs generated per each program, we also take note of the main challenges that require manual modifications. The results reveal potential opportunities for future improvement in CHEKPROP.

To assess the effectiveness of generated PBTs, we study if it checks the property over representatives of all or most partitions of the input space. As explained in Sect. 2, one of the main components of a PBT is an input generator that produces various inputs from the input space. In this experiment, we determine to what extent the input generators of generated PBTs produce inputs from all partitions of the input space. The more partitions of input space considered by a PBT, the more effective the PBT is. We assess the effectiveness of PBTs generated for each program through a manual analysis and assign them to one of the three effective groups "HIGH", "MED", and "LOW".

II. Results: Table 3 summarizes the result of our experiment on CHEKPROP applicability. The "Property_ID" and "Program" columns indicate the property and the program that the PBT is testing. Note that the ID of the property is taken from Table 2, which lists the properties extracted by CHEKPROP. The "Executability" column shows the result of our assessment of the executability of generated PBTs in terms of their syntactical and semantical correctness. The fourth column presents the main executability challenge of generated PBTs that should be addressed manually. Finally, the last column presents the level of effectiveness of PBTs generated for each program.

For 47% (10/21) of the relevant properties (Pr1, Pr2, Pr3, Pr4, Pr5, Pr6, Pr7, Pr8, Pr9, and Pr17), the generated PBTs are executable without major changes that require less than one minute of manual work. In fact, for none of these PBTs, except for the Pr17 PBT, no major executability issues are detected. These PBTs successfully execute and pass with little to none manual modification. Also, for Pr17 PBT, the problem is that the generated PBT runs the program for too many inputs, leading to a timeout. A developer who knows the logic of Pr17 property of the Remote Buggy program can fix the generated PBT by only modifying the number of random inputs that should be considered. Given the complexity of predicting the time needed for running a test on a CPS, this case shows the importance and positive impact of keeping a human in the loop of LLM-based PBT generation. In sum, our analysis of the executability of PBTs generated for Pr1, Pr2, Pr3, Pr5, Pr6, Pr7, Pr9, and Pr17 shows that for a remarkable number of relevant CPS properties CHEKPROP generates a PBT that is executable with minor manual modifications.

For seven of the properties (Pr14, Pr15, Pr20, Pr21, Pr23, Pr24, and Pr26), the only main challenge to executability of generated PBTs occurs in their mocking of the CPS. This challenge occurs because the mocking method employed has a conflict with property-based testing of the CPS under test. In particular, every time the test is executed for a specific input, all the pins used in the mock of the CPS should be initialized from scratch. However, the mocking method used in these PBTs only initializes the mock object once for all inputs considered in the PBT. This leads to a semantic problem with the logic of the CPS under test, as well as a syntactic error in using the `hypothesis` library. Consequently, these seven PBTs require a medium level of manual modification to become executable.

Table 3. The quality of PBTs generated by CHEKPROP for the 21 relevant propertie. A "HIGH", "MED", or "LOW" level in the "Executability" column indicates the PBT can be successfully executed with less than one minute, between one to three minutes, or more than three minutes of manual effort for modification, respectively. The "Effectiveness" column indicates the level of input space partitions covered by the PBT.

Property_ID	Program	Executability	Main Executability Challenge	Effectiveness
Pr1	TCS	HIGH	No major executability issues detected.	HIGH
Pr2	TCS	HIGH	No major executability issues detected.	HIGH
Pr3	TCS	HIGH	No major executability issues detected.	HIGH
Pr4	TCS	HIGH	No major executability issues detected.	HIGH
Pr5	PCS	HIGH	No major executability issues detected.	HIGH
Pr6	PCS	HIGH	No major executability issues detected.	HIGH
Pr7	PCS	HIGH	No major executability issues detected.	HIGH
Pr8	PCS	HIGH	No major executability issues detected.	HIGH
Pr9	Laser Tripwire	HIGH	No major executability issues detected.	MED
Pr10	Line Following Robot	LOW	Wrong parameter passed to the `pin.drive_up()` method.	HIGH
Pr11	Line Following Robot	LOW	Wrong parameter passed to the `pin.drive_up()` method.	HIGH
Pr12	Line Following Robot	LOW	Wrong parameter passed to the `pin.drive_up()` method.	HIGH
Pr14	Ultrasonic Thermin	MED	The used mocking method does not work for parameterized tests.	HIGH
Pr15	Ultrasonic Thermin	MED	The used mocking method does not work for parameterized tests.	HIGH
Pr17	Remote Buggy	HIGH	Exceeds timeout as tested on too many inputs.	HIGH
Pr20	Presence Indicator	MED	The used mocking method does not work for parameterized tests.	MED
Pr21	Presence Indicator	MED	The used mocking method does not work for parameterized tests.	MED
Pr23	GPIOZERO InputDevice	MED	The used mocking method does not work for parameterized tests.	HIGH
Pr24	GPIOZERO InputDevice	MED	The used mocking method does not work for parameterized tests.	HIGH
Pr25	GPIOZERO InputDevice	LOW	The used mocking method does not work for parameterized tests. The pin state is set with an incorrect use of the interface.	HIGH
Pr26	GPIOZERO InputDevice	MED	The used mocking method does not work for parameterized tests.	HIGH

We notice that mocking CPS is both tricky and essential for testing. As CPSs are supposed to run in a physical environment, we need to mock how the environment affects CPS programs. This can require a detailed understanding of the relationship between various components of given CPS programs. Previous work shows that such domain knowledge can be effectively provided to LLMs by in-context learning, i.e., adding relevant examples to the prompt [7]. Based on this observation, we suggest that practitioners use a few-shot prompt with mocking examples to generate PBTs for cyber-physical systems with LLMs.

Finally, one of the main issues with generated PBTs for four properties (Pr10, Pr11, Pr12, and Pr25) is how they use GPIOZERO. For example, the PBT generated for Pr25 uses output pins of the Raspberry Pi board to initialize their object of the `InputDevice`. With further analysis, we realize that fixing the issues in this PBT depends on a deep understanding of multiple GPIOZERO classes. However, the current version of CHEKPROP only includes the documentation of the `InputDevice` class in the prompt. This documentation is taken from the comments presented in Listing 4. To fix the issue with this PBT, the LLM also needs to have the documentation for other classes, such as `PiGPIOFactory`. We conclude that a strong LLM-based PBT generation tool for CPS programs requires augmenting prompts with all relevant information from the program documents.

Our analysis of the effectiveness of the generated PBTs shows that the PBTs generated for 85% (18/21) of the properties are highly effective; these PBTs test the property on most partitions of the input space. With a more detailed look, we observe that the generated tests tend to be more effective when the tests employ a straightforward and flexible mock of the CPS components. For example, as shown in Listing 3, our Python implementation of TCS provides a `MockRoom` class. This class enables a tester to run the program with many different inputs only by changing a few parameters regarding the starting temperature of the room and the timing of updating various components. Using this mock class, the generated PBTs for TCS properties run the program with different configurations that represent all partitions of the input space. This experiment also reaffirms the importance of using flexible mocks with a straightforward API for testing CPSs.

Answer to RQ2: What is the quality of CHEKPROP generated PBTs for real-world CPSs?
We assess the applicability of PBTs generated by CHEKPROP 21 relevant properties in our dataset from two aspects: executability and effectiveness. Our results reveal that a remarkable number of the generated PBTs are highly executable (47%) and highly effective (85%), which indicates the applicability of CHEKPROP. Our analysis also leads to two major suggestions for generating high-quality PBTs for CPSs with LLMs. First, the LLM should be aided in employing straightforward and flexible mocking by providing well-designed few-shot examples in the prompt. Secondly, it is important to include the relevant documentation from all parts of the CPS in the prompt. These two techniques make our proposed approach even more robust and practical.

5 Related Work

The application of LLMs to test CPS is at early stages and many of the efforts have been focused on scenario generation for autonomous driving and robotics. For instance, OmniTester [11] uses an LLM (GPT-4) to generate diverse driving scenarios from natural language descriptions and proposes test road layouts and events. They also incorporates retrieval-augmented generation (RAG) and iterative self-improvement to refine scenarios. Petrovic et al. [14] similarly incorporates LLMs into an autonomous vehicle testing pipeline. Their approach provides the LLM with a formal environment model (metamodel of roads, vehicles, pedestrians, etc.) and standardized requirements as context. The LLM is prompted to produce a concrete test scenario (in a JSON format executable in the CARLA simulator) that satisfies the given requirements. They use the LLM to translate natural language requirements into Object Constraint Language (OCL) rules—formalizes expected environmental and safety properties. The OCL properties are then checked against the generated test scenario and if required, the feedback is sent to the LLM for correction before the execution of the test scenario. Besides automotive, other related works are emerging in robotics. For example, Wang et al. [24] show that GPT-4 can automatically generate robotic simulation tasks (including environment configurations and goals). They mainly address test scenario generation (test environments and test inputs) rather than directly inferring formal properties or invariants from system specifications. They show that LLMs can handle the environmental context of CPS testing when guided by domain models.

In the broader software systems context, LLMs have been utilized for automated test case generation from various sources of specification. Many of the approaches target conventional software systems (without either ML or any physical components) and have shown promising results in automating unit test creation. Kang et al. [10] present LIBRO, a framework that uses an LLM to generate JUnit tests from bug reports. The goal is to reproduce reported defects automatically as the conventional test generators generally struggle with understanding the semantic intent of a bug report. LIBRO's performance evaluation on the Defects4J benchmark found that it can produce failing tests for about 33% of bugs and demonstrates that an LLM can interpret natural language bug descriptions and translate them into fault-revealing code. Another set of work explores using LLMs to generate tests from requirement documents or user stories. Rahman and Zhu [15], leverage GPT-4 to produce test-case specifications (in JSON) directly from high-level requirements and intend to bridge the gap between specifications and executable tests. Some approaches also utilize LLMs within an interactive test generation process. Chen et al. [3] introduce ChatUniTest, an LLM-based unit test generation framework. In their approach, the LLM (Code Llama) drafts a Java unit test; the test is executed to see if it passes or if it exercises the intended code; any errors or unsatisfied goals are fed back for the LLM to repair and refine the test.

Alshahwan et al. [2], report using LLMs to extend and improve existing test suites in an industrial setup—focuses on corner-case inputs that developers missed. Their tool generates additional unit tests to increase coverage of tricky edge conditions. Overall, surveys of the field, e.g., Wang et al., 2024 [23] conclude that LLMs show strong potential in automated testing by reducing the manual effort to write test cases—mainly in code-centric contexts such as unit testing.

Regarding Property-based testing with LLMs, applying LLMs to the generation of PBTs has recently emerged. The most relevant work is by Vikram et al. (2024), which investigates if LLMs write good PBTs [22]. They investigate using GPT-4 and other models to automatically generate PBT code (using the Hypothesis framework in Python) from API documentation. In their setup, the LLM is given the documentation of a library function in natural language and prompted to generate a property-based test. The generated test produces appropriate random inputs and asserts the documented properties on the outputs. They evaluate the validity (the test must run without errors), soundness (the test assertions should hold for correct implementations and fail for buggy ones) and property coverage (how many distinct expected properties are captured by the test) of the tests.

CPS Challenges and our Contribution: The works mentioned above establish a foundation for the generation of LLM-driven property-based tests for CPS, which can also act as a complement to other safety assurance approaches like verification-based development techniques [18]. Those approaches are often limited to the abstraction captured by the model and face scalability challenges with complex, real-world scenarios. Property-based testing can address the limitations by generating diverse and extensive test cases that can uncover defects and implementation errors or environmental interactions not represented explicitly in formal models, then, acting as an empirical validation layer. In this context, in prior studies like Vikram et al.'s [22], the system under test is a software API with no external physical connected components and the LLM did not need to reason about sensors, actuators, or continuous dynamics. But in a cyber-physical system, properties often relate to the interaction between software and the physical components, which are more complex to formalize and test. Environmental mocking becomes a necessity—a model or simulation of the physical environment is required to represent the real world.

Recent CPS testing approaches with LLMs (e.g. for autonomous driving [14]) addressed this by restricting the LLM to consider a domain metamodel and produce output in a structured format for a simulator. This helps ensure some basic physical realism in generated scenarios, but it does not guarantee that all relevant properties can be identified or verified. Our approach supports testing and also runtime property-based monitoring. This means the LLM is used to derive property assertions that can also run alongside the deployed system, to check for violations in the runtime. Our approach extends the frontier by applying LLM-driven property-based test generation to CPS, in which both the inference of the generated properties from code documentation and the execution of the corresponding tests must account for the intended CPS programs under test.

6 Conclusion

In this paper, we propose a novel approach for automatically guardrailing cyber-physical system. This approach employs LLMs to generate property-based tests for CPS programs that can be used to monitor CPSs behavior and detect unsafe states. We implement a prototype of this approach in ChekProp and evaluate it on real-world and commonly studied CPSs. We find that ChekProp is applicable on real-world CPSs. More specifically, ChekProp extracts relevant properties, comparable to manually crafted properties, and then generates executable and effective property-based tests that verify these properties.

Our experiments reveal two major challenges for LLM-enabled property-based test generation for CPSs and suggest potential solutions to these challenges. First, given the limited number of public CPS projects, LLMs are not trained on a vast dataset of CPS source code. Consequently, LLMs generated tests may not correctly capture the relation between the API of CPS program and the physical status of the system. For example, while the LLM might recognize that verifying a specific property, like keeping the temperature in a certain range, requires activating the room's heater, it may not identify the correct method in the CPS code to effect this change. To address this issue, the prompts should contain relevant documents and source code taken from all parts of the project under test. This helps the LLM better understand the API of the CPS code and make the correct use of it.

The second main challenge that we observe is the complexities involved in mocking CPSs. In cyber-physical systems, we deal with the cyber part, the physical part, and the environment. PBTs should run at different times and test the system under various environmental conditions. This requires mocking inputs sometimes from the environment and sometimes from the physical system, and sometimes both. Such mocking demands a correct understanding of the specific environmental inputs to the CPS, their interrelations, and the interface between the cyber and physical components. Our experiments show that LLMs often struggle to capture such an understanding, leading to incorrect generation of PBTs. To address this issue, we suggest including extensive examples of proper mocking scenarios for the CPS under test in the prompt. Prior studies show that such examples can significantly improve the ability of LLMs to generate correct tests [10]. In summary, our study is a solid first step in creating guardrails for CPSs with LLM-generated property-based tests and proves the promising future of this research path.

Acknowledgment. This work is supported by the Trusted Smart Systems pre-study grant at Mälardalen University, as well as Sweden's Innovation Agency Vinnova through the INTelligent sEcuRity SoluTIons for Connected vEhicles (INTERSTICE) project with grant number 2024-00661 and the Flexible and Secure Modular Automation (FLEXATION) project with grant number 2024-01731.

Appendix A: Example Prompts and Generated PBTs

Here we present two examples of the prompts that CHEKPROP sends to the LLM and a selected subset of PBTs that are generated based on these prompts (Figs. 4 and 5).

Listing 5. A PBT generated by CHEKPROP for the line following robot program.

```
@given(
    left_sensor_value=st.integers(min_value=0, max_value=1),
    right_sensor_value=st.integers(min_value=0, max_value=1),
    speed=st.floats(min_value=0.1, max_value=1.0)
)
def test_motor_control_based_on_sensor_values(left_sensor_value, right_sensor_value,
     speed):
    Device.pin_factory = MockFactory()
    with LineFollowingRobot(Motor(2, 3, enable=4, pwm=False), Motor(5, 6, enable=7,
     pwm=False), ...) as lfr:
        if left_sensor_value > 0:
            lfr.left_sensor.pin.drive_high()
        if right_sensor_value > 0:
            lfr.right_sensor.pin.drive_high()
        time.sleep(0.1) # Simulate slight delay to trigger motor_speed updates

        if left_sensor_value == 0 and right_sensor_value == 0:
            assert lfr.right_motor.value == speed
            assert lfr.left_motor.value == speed
        elif left_sensor_value == 0 and right_sensor_value == 1:
            assert lfr.left_motor.value == -speed
        elif left_sensor_value == 1 and right_sensor_value == 0:
            assert lfr.right_motor.value == -speed
        else:
            assert lfr.left_motor.value != 0 or lfr.right_motor.value != 0
```

Listing 6. Two PBTs generated by CHEKPROP for the *InputDevice* class.

```
@given(st.booleans())
def test_close_releases_pin(pull_up):
    Device.pin_factory = MockFactory(pin_class=MockPin)
    device = InputDevice(4, pull_up=pull_up)
    device.close()
    assert device.pin is None

@given(st.booleans())
def test_active_state_affects_is_active_basic(pull_up):
    Device.pin_factory = MockFactory(pin_class=MockPin)
    with InputDevice(4, pull_up=pull_up) as device:
        device.pin.drive_high()
        assert device.is_active == (not pull_up)
        device.pin.drive_low()
        assert device.is_active == pull_up
```

~~~~
The following is a description of a line following robot.

{Natural Language Description}
--------------------------------------------------------------------------------
The following code implements this line following robot using the gpiozero library for Raspberry Pi programs. You should generate property based tests for this code.

```python
class LineFollowingRobot(Robot):
    def __init__(self, left, right, left_seonsor_pin=17, right_sensor_pin=27, speed=0.65):
        super().__init__(left, right)
        self.left_sensor = LineSensor(left_seonsor_pin)
        self.right_sensor = LineSensor(right_sensor_pin)
        self.speed = speed
        self.source = self.motor_speed()

    def motor_speed(self):
        left_mot = 0
        right_mot = 0
        while True:
            left_detect  = int(self.left_sensor.value)
            right_detect = int(self.right_sensor.value)
            # Stage 1
            if left_detect == 0 and right_detect == 0:
                left_mot = 1
                right_mot = 1
            # Stage 2
            if left_detect == 0 and right_detect == 1:
                left_mot = -1
            if left_detect == 1 and right_detect == 0:
                right_mot = -1
            print(right_mot, left_mot)
            yield right_mot * self.speed, left_mot * self.speed

    def stop(self):
        self.robot.stop()
        self.left_sensor.close()
        self.right_sensor.close()
        super().stop()

```
--------------------------------------------------------------------------------
The following is a unit test for this system class:

```python

def test_left_sensor_activation_moves_right_motor():
    # Create a MockFactory and use it as pin_factory on Device
    Device.pin_factory = MockFactory()

    with LineFollowingRobot(Motor(2, 3, enable=4, pwm=False),
                            Motor(5, 6, enable=7, pwm=False),
                            left_seonsor_pin=8, right_sensor_pin=9,
                            speed=1) as lfr:

        time.sleep(1)
        lfr.left_sensor.pin.drive_high()
        time.sleep(1)
        assert lfr.right_motor.value == 1
```
--------------------------------------------------------------------------------
Generate property based tests for this system following the steps below:
    1. Based on the given description and code, extract the properties of the system.
    2. Use the unit tests to understand the behavior and interface of the code.
    3. Based on the extracted properties and your understanding of the code, use the hypothesis
       library to generate property based tests.
~~~~

Fig. 4. The prompt used by CHEKPROP to generate PBTs for the line following robot program.

````
The following is the description of the InputDevice class in the gpiozero library.

"""
InputDevice class represents a generic GPIO input device.This class extends :class:`GPIODevice` to add facilities common to GPIO input devices. The constructor adds the optional *pull_up* parameter to specify how the pin should be pulled by the internal resistors. The :attr:`is_active` property is adjusted accordingly so that :data:`True` still means active regardless of the *pull_up* setting.

:type pin: int or str
:param pin:
        The GPIO pin that the device is connected to. See :ref:`pin-numbering` for valid pin
        numbers. If this is :data:`None` a :exc:`GPIODeviceError` will be raised. :type pull_up:
        bool or None :param pull_up: If :data:`True`, the pin will be pulled high with an internal
        resistor. If :data:`False` (the default), the pin will be pulled low.  If :data:`None`, the
        pin will be floating. As gpiozero cannot automatically guess the active state when not
        pulling the pin, the *active_state* parameter must be passed.

:type active_state: bool or None
:param active_state:
        If :data:`True`, when the hardware pin state is ``HIGH``, the software pin is ``HIGH``. If
        :data:`False`, the input polarity is reversed: when the hardware pin state is ``HIGH``, the
        software pin state is ``LOW``. Use this parameter to set the active state of the underlying
        pin when configuring it as not pulled (when *pull_up* is :data:`None`). When *pull_up* is
        :data:`True` or :data:`False`, the active state is automatically set to the proper value.

:type pin_factory: Factory or None
:param pin_factory:
        See :doc:`api_pins` for more information (this is an advanced feature
        which most users can ignore).
"""
------------------------------------------------------------------------------------------------
The following code is the implemention of the InputDevice class of the gpiozero library.

```python

class InputDevice(GPIODevice):
    def __init__(self, pin=None, *, pull_up=False, active_state=None,
                 pin_factory=None):
        super().__init__(pin, pin_factory=pin_factory)
        try:
            self.pin.function = 'input'
            pull = {None: 'floating', True: 'up', False: 'down'}[pull_up]
            if self.pin.pull != pull:
                self.pin.pull = pull
        except:
            self.close()
            raise

        if pull_up is None:
            if active_state is None:
                raise PinInvalidState(
                    f'Pin {self.pin.info.name} is defined as floating, but '
                    f'"active_state" is not defined')
            self._active_state = bool(active_state)
        else:
            if active_state is not None:
                raise PinInvalidState(
                    f'Pin {self.pin.info.name} is not floating, but '
                    f'"active_state" is not None')
            self._active_state = False if pull_up else True
        self._inactive_state = not self._active_state

    @property
    def pull_up(self):
        """
        If :data:`True`, the device uses a pull-up resistor to set the GPIO pin
        "high" by default.
        """
        pull = self.pin.pull
        if pull == 'floating':
            return None
        else:
            return pull == 'up'

```
------------------------------------------------------------------------------------------------
The following is a unit test for this class:

```python

def test_input_initial_values():
    # Create a MockFactory and use it as pin_factory on Device
    Device.pin_factory = MockFactory()
    with InputDevice(4, pull_up=True) as device:
        assert repr(device).startswith('<gpiozero.InputDevice object')
        assert device.pin.function == 'input'
        assert device.pin.pull == 'up'
        assert device.pull_up
    assert repr(device) == '<gpiozero.InputDevice object closed>'
    with InputDevice(4, pull_up=False) as device:
        assert device.pin.pull == 'down'
        assert not device.pull_up
```
------------------------------------------------------------------------------------------------
Generate property based tests for this system following the steps below:
    1. Based on the given description and code, extract the properties of the system.
    2. Use the unit tests to understand the behavior and interface of the code.
    3. Based on the extracted properties and your understanding of the code, use the hypothesis library to generate property based
       tests.
````

Fig. 5. The prompt used by CHEKPROP to generate PBTs for the *InputDevice* class.

References

1. Abshari, D., Fu, C., Sridhar, M.: LLM-assisted physical invariant extraction for cyber-physical systems anomaly detection. arXiv preprint arXiv:2411.10918 (2024)
2. Alshahwan, N., et al.: Automated unit test improvement using large language models at meta. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pp. 185–196 (2024)
3. Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., Yin, J.: ChatUniTest: a framework for LLM-based test generation. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pp. 572–576 (2024)
4. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, pp. 268–279 (2000)
5. Etemadi, K., et al.: Chekprop (2025). https://github.com/khesoem/ChekProp
6. Fink, G., Bishop, M.: Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Softw. Eng. Notes **22**(4), 74–80 (1997)
7. Giagnorio, A., Martin-Lopez, A., Bavota, G.: Enhancing code generation for low-resource languages: no silver bullet. arXiv preprint arXiv:2501.19085 (2025)
8. ISO 25000: ISO/IEC 25010 - System and software quality models (2019). https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
9. Jiang, J., Wang, F., Shen, J., Kim, S., Kim, S.: A survey on large language models for code generation. arXiv preprint arXiv:2406.00515 (2024)
10. Kang, S., Yoon, J., Yoo, S.: Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 2312–2323. IEEE (2023)
11. Lu, Q., Wang, X., Jiang, Y., Zhao, G., Ma, M., Feng, S.: Multimodal large language model driven scenario testing for autonomous vehicles. arXiv preprint arXiv:2409.06450 (2024)
12. MacIver, D.R., Hatfield-Dodds, Z., et al.: Hypothesis: a new approach to property-based testing. J. Open Source Softw. **4**(43), 1891 (2019)
13. Moradi, F., Asadollah, S.A., Pourvatan, B., Moezkarimi, Z., Sirjani, M.: Crystal framework: cybersecurity assurance for cyber-physical systems. J. Log. Algebr. Methods Program. **139**, 100965 (2024)
14. Petrovic, N., et al.: LLM-driven testing for autonomous driving scenarios. In: 2024 2nd International Conference on Foundation and Large Language Models (FLLM), pp. 173–178. IEEE (2024)
15. Rahman, T., Zhu, Y.: Automated user story generation with test case specification using large language model. arXiv preprint arXiv:2404.01558 (2024)
16. Raspberry Pi Team: Raspberry Pi Project Selector (2025). https://projects.raspberrypi.org/en/projects
17. Sirjani, M., Khamespanah, E.: On time actors. Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday, pp. 373–392 (2016)
18. Sirjani, M., Provenzano, L., Asadollah, S.A., Moghadam, M.H., Saadatmand, M.: Towards a verification-driven iterative development of software for safety-critical cyber-physical systems. J. Internet Serv. Appl. **12**(1), 2 (2021)
19. gpiozero Team: A simple interface to gpio devices with Raspberry Pi (2025). https://github.com/gpiozero/gpiozero
20. Tillmann, N., Schulte, W.: Parameterized unit tests. ACM SIGSOFT Softw. Eng. Notes **30**(5), 253–262 (2005)

21. Tiwari, D., Gamage, Y., Monperrus, M., Baudry, B.: Proze: generating parameterized unit tests informed by runtime data. In: 2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM), pp. 166–176. IEEE (2024)
22. Vikram, V., Lemieux, C., Sunshine, J., Padhye, R.: Can large language models write good property-based tests? arXiv preprint arXiv:2307.04346 (2023)
23. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., Wang, Q.: Software testing with large language models: survey, landscape, and vision. IEEE Trans. Softw. Eng. (2024)
24. Wang, L., et al.: GenSim: generating robotic simulation tasks via large language models. arXiv preprint arXiv:2310.01361 (2023)

RAG and Agentic Assistant: A Combined Approach

Moez Ben Hajhmida[1(✉)] and Edward A. Lee[2]

[1] University of Tunis El Manar, Tunis, Tunisia
moez.benhajhmida@enit.utm.tn
[2] University of California, Berkeley, Berkeley, USA
eal@berkeley.edu

Abstract. While large language models (LLMs) have exhibited strong capabilities in translating code, particularly from C to Python, their performance noticeably declines when dealing with less common languages like LF. Prior research indicates that Retrieval-Augmented Generation (RAG) can enhance LLMs capabilities in code generation by integrating codebase retrieval. Despite its promise, RAG systems are constrained by LLMs capabilities to deal with less common languages. Agentic AI coding assistants offer a different approach by acting as AI co-developers, automating tedious tasks and allowing developers to focus on high-level design. This paper proposes a novel system that combines RAG with agentic AI assistants to improve the accuracy of converting LF programs with target C into LF code with target Python. We conduct a comparative evaluation of state-of-the-art proprietary and open-source code LLMs in this task, demonstrating that RAG can significantly narrow the performance gap between small and large language models. Furthermore, we integrate an agentic assistant within an AI-powered IDE to automate developer-assisted error correction and refactoring, streamlining the development workflow. In terms of syntax correctness and successful execution rates, experiments highlight the significant improvements achieved by the combined approach.

1 Introduction

Large Language Models (LLMs) represent a significant leap forward in software development automation, exhibiting impressive abilities in both code generation and translation. However, a significant performance gap exists when these models encounter less commonly used programming languages, a challenge amplified in the context of Domain-Specific Languages (DSLs) [2,16]. One such language is Lingua Franca (LF) [11], a coordination language for real-time distributed systems. Currently, the Lingua Franca project has 150 Lingua Franca test files written for the C target that haven't been implemented in Python yet. To ensure comprehensive testing, we need to address the disparity between the number of C and Python Lingua Franca (LF) test files. Thus, our immediate goal is to

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 47–62, 2026.
https://doi.org/10.1007/978-3-032-07132-3_4

develop and implement the missing Python test files to achieve parity with the C test suite.

To address the code generation performance gap, we investigate whether augmenting LLMs with external contextual information via Retrieval-Augmented Generation (RAG) [19] can enhance their translation performance for LF code. RAG extends LLMs by retrieving relevant knowledge units from a curated database and injecting them into the generation prompt, enabling more informed outputs without the need for extensive model fine-tuning.

Additionally, code translation tasks often require iterative corrections to address syntax errors, language-specific constructs, or semantic inconsistencies. Traditional LLM-based workflows rely heavily on human oversight for these adjustments. In contrast, agentic AI systems [3], which can autonomously perform sequences of steps, offer a promising avenue for automating these corrective processes. Agentic AI coding assistants offer a collaborative, developer-assisted approach by automating routine coding tasks, enabling developers to concentrate on higher-level design.

In this paper, we propose a hybrid approach combining RAG-based code translation with an agentic AI assistant integrated into an AI-powered IDE. We conduct empirical experiments translating LF programs with target C into comparable programs with target Python and evaluate a range of code LLMs with and without RAG augmentation. The experiments are conducted on 150 regression test programs that were manually written for the C target but where there are no comparable regression tests for the Python target. Furthermore, we deploy the Cursor IDE's agentic assistant to iteratively refactor, validate, and standardize the generated code. Our results show that combining RAG and agentic assistants significantly improves both translation success rates and workflow efficiency, even with comparatively small LLMs.

This paper is organized as follows: Sect. 2 provides an overview of the fundamentals of Lingua Franca language. Section 3 details a comparative assessment of contemporary proprietary and open-source code LLMs in the context of code translation, providing evidence that RAG can narrow the performance gap between smaller and larger models. In Sect. 4, we discuss how RAG and Agentic AI systems can enhance code LLM capabilities. Section 5 introduces our proposed approach, combining the RAG technique with agentic AI coding assistant to enhance LF code translation. Subsequently, we summarize the findings of our experimental evaluations translating 150 LF files. Finally, Sect. 7 concludes the paper.

We conduct a comparative evaluation of state-of-the-art proprietary and open-source code LLMs in this task, demonstrating that RAG can significantly narrow the performance gap between small and large models. Furthermore, we integrate an agentic assistant within an AI-powered IDE to automate developer-assisted error correction and refactoring, streamlining the development workflow. In terms of syntax correctness and successful execution rates, the experimental results highlight the significant improvements achieved by the combined approach.

2 Lingua Franca

Lingua Franca (LF) [11] is an open-source domain-specific language for building high-performance, secure, and reliable distributed real-time systems. LF is a polyglot coordination language that facilitates the development of distributed applications by employing a reactor-based architecture. *Reactors* are reactive components programmed in popular programming languages like C/C++, Python, Rust, and TypeScript.

An LF application is made of *reactors* connected together with ports and connections. Reactors are deterministic actors whose behavior is specified through *reactions*. Reactions are triggered by discrete events fired at specific logical time instants. LF is supported by a runtime system that facilitates communication between connected reactors, ensuring predictable and consistent execution, even in distributed environments. Figure 1 showcases an example of a Ligua Franca code for C target with three reactors: Source, Destination, and a main reactor. In this example, the main reactor has a single reaction, which is triggered by the startup trigger. This trigger causes the reaction to execute at the start of the program. The body of the reaction, delimited by {= ... =}, is ordinary C code. More elaborate examples can be found at https://github.com/lf-lang/playground-lingua-franca.

```
target C

reactor Source {
  output out: int

  reaction(startup) -> out {=
    lf_set(out, 42);
  =}
}

reactor Destination {
  input in: int

  reaction(in) {=
    interval_t time = lf_time_logical_elapsed();
    printf("Received %d at logical time " PRINTF_TIME ".\n", in->value, time
        );
    if (time <= 0LL) {
      fprintf(stderr, "ERROR: Logical time should have been greater than
          zero.\n");
      exit(1);
    }
  =}
}

main reactor {
  source = new Source()
  destination = new Destination()
  source.out -> destination.in
}
```

Fig. 1. Example of a LF code for C target with one reactor.

Analogous to object-oriented programming, a developer declares the reactors involved in the application and specifies their interactions using the LF-specific language. The logic for each reactor is then defined through a set of reactions. Each reaction's implementation is written in a target language, such as C/C++, Python, Rust, or TypeScript. Proficient Python programmers unfamiliar with LF may struggle to learn the LF syntax. Consequently, we conducted experiments to evaluate the feasibility of using Large Language Models (LLMs) for automating the generation of LF programs.

3 Code LLMs

Code LLMs (Large Language Models for programming) are AI models specifically trained on extensive programming datasets spanning multiple languages, including both source code and related text like documentation. This specialized training allows them to excel at processing and generating code with high accuracy. Their capabilities include automatically completing code segments, converting natural language specifications into functional code, translating between programming languages, identifying and fixing bugs, and providing clear explanations of code functionality. Built on advanced deep learning architectures, these models are increasingly valuable tools that enhance developer workflows and can significantly improve programming efficiency.

Leading proprietary large language models like GPT-4 [17] and Claude [1] have exhibited impressive performance, showcasing advanced abilities in language understanding, reasoning, planning, and code generation. On the HumanEval benchmark [4], the widely recognized standard for assessing code generation capabilities, GPT-4 achieved a Pass@1 score of 87.8% [15], while Claude attained 84.9% [1].

While proprietary models have set a high bar, the landscape of code LLMs also features increasingly powerful open-source alternatives. Projects like Llama[1], DeepSeek[2], and others are rapidly closing the performance gap, offering researchers and developers greater flexibility and control. These open-source models are often fine-tuned and adapted by the community, leading to rapid innovation and specialized versions that can excel in specific coding tasks or programming languages. The performance of open-source code LLMs on the HumanEval benchmark has seen remarkable improvements (approaching 87%), as illustrated by the BigCode models leaderboard[3], which showcases the evolution of remarkable models like Starcoder2 [12], Qwen2.5-Coder [9], CodeLlama [21], DeepSeek-coder [6], and Codegeex2 [25].

OpenAI's **GPT-4**, a leading large multimodal model announced in March 2023, excels in understanding and generating text and images. While specific architectural details and parameter counts remain proprietary, GPT-4 has demonstrated significant improvements over its predecessors in areas such as reasoning, complex instruction following, and creative content generation. Initially achieving a 67.0% Pass@1 score on the HumanEval code generation benchmark

[1] https://www.llama.com/.
[2] https://www.deepseek.com/.
[3] https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard.

[17], its performance has since been enhanced to 87.8% through further development [15], including iterations like GPT-4o.

Anthropic's **Claude-3** model family, launched in March 2024, features a trio of large multimodal models tailored for different needs: the top-performing Opus, the balanced Sonnet, and the swift Haiku. Evaluating their zero-shot Python coding abilities on the HumanEval benchmark, these models achieved the following Pass@1 scores respectively: Opus at 84.9%, Sonnet at 73.0%, and Haiku at 75.9% [1].

Announced in February 2024, **StarCoder-2** is a family of open-access large language models for code generation, developed by the Hugging Face-supported BigCode[4] project. It comes in three sizes: 3B, 7B, and 15B parameters, all trained on over 600 programming languages from The Stack v2 dataset. Key architectural features include Grouped Query Attention and a large context window of 16,384 tokens. On the HumanEval benchmark, the StarCoder 2 fine-tuned models achieved the following Pass@1 scores: the 3B model reached 45.12%, the 7B model scored 51.22%, and the 15B model attained 59.15%. These results position the smaller StarCoder 2 models as competitive within their size range, with the 15B model demonstrating strong performance [12].

Released around late 2024 by Alibaba Cloud's Qwen team[5], **Code-Qwen 2.5** is a series of open-source code LLMs ranging from 0.5B to 32B parameters. These models emphasize improved code generation with a larger context of 128K tokens [9]. The 32B instruct model reportedly achieves state-of-the-art open-source performance in coding tasks, demonstrating capabilities comparable to GPT-4o with a Pass@1 score of 83.2%[6].

Announced by Meta in August 2023, **Code Llama-2** is an open-source family of large language models specifically designed for code-related tasks. Built upon the Llama 2 architecture, it's available in sizes from 7B to 70B parameters, with specialized versions for Python and instruction following. It has demonstrated strong performance on code benchmarks like HumanEval, with the 70B instruct model achieving around 67.8% Pass@1 [21].

CodeGeeX-2, developed by Tsinghua University and announced in July 2023, is a 6-billion parameter multilingual code generation model. Based on the ChatGLM2 architecture [5], it has shown significant improvements over its predecessor, CodeGeeX. Notably, it has demonstrated strong performance on the HumanEval benchmark across multiple programming languages, achieving a Pass@1 score of 35.9% [25].

DeepSeek-Coder-2, developed by DeepSeek AI, is a series of open-source code language models announced in late 2023 (DeepSeek-Coder-1) and early 2024 (DeepSeek-Coder-V2). It comes in various sizes, ranging from 1.3B to 33B parameters, and was trained from scratch on a massive dataset composed of 60% source code, 10% math corpus, and 30% natural language corpus.

---

4 https://www.bigcode-project.org/.

5 https://www.alibabacloud.com/en/solutions/generative-ai/qwen.

6 https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard.

The instruction-tuned 33B DeepSeek-Coder-V2 model achieved 90.2% on the HumanEval benchmark, while smallest model achieved 37.2% [6].

Table 1 provides a comparative overview of several prominent code LLMs with a strong focus on code-related capabilities. It highlights key characteristics such as the developing institution, model size (in parameters), vocabulary size (in tokens), and context window length. The context window of an LLM refers to the maximum amount of text (measured in tokens) that the model can consider when processing an input and generating an output. Given that OpenAI and Anthropic do not publicly disclose the exact model size and vocabulary size for GPT-4 and Claude 3, the values presented for these parameters are estimations. This information allows for a quick comparison of these leading models in terms of their scale and context handling abilities.

Table 1. Overview of Code Large Language Models

Model	Institution	Size	Vocabulary	Context Window
GPT-4 [17]	OpenAI	∼200B	∼200k	32K
Claude-3 [1]	Anthropic	∼175B	∼200k	200K
StarCoder-2 [12]	Hugging Face	3B, 7B, 15B	49k	16k
Code-Qwen-2.5 [9]	Alibaba	7B	92k	128k
Code Llama-2 [21]	Meta	7B, 13B, 34B	32k	16k
CodeGeeX-2 [25]	Tsinghua	6B	65k	8k
DeepSeek-Coder-2 [6]	DeepSeek	1.3B, 6.7B, 33B	32k	16k

Pan et al. [18] conducted an empirical study assessing the potential and limitations of LLMs in code translation. Their evaluation involved translating 800 CodeNet [20] samples (200 for each of C++, Go, Java, and Python) and reported the following success rates for various models: GPT-4 (83.0%), StarCoder (42.0%), Llama 2 (14.9%), and CodeGeeX (14.9%). Based on their findings, Pan et al. suggest that enriching the context provided to LLMs during code translation can lead to improved output. To explore this, they introduce a prompt-engineering method informed by common translation errors, which resulted in an average performance gain of 5.5% for LLM-based code translation. The authors highlight that providing more context to LLMs during translation can help them produce better results.

In a recent empirical study, Macedo et al. [14] analyzed the output of eleven widely used instruction-tuned LLMs (ranging from 1B to 46.7B parameters) on 3,820 translation examples across C, C++, Go, Java, and Python. Their findings indicate that 26.4% to 73.7% of the generated translations require post-processing due to the inclusion of non-code elements like quotes and text. They further demonstrate that a well-designed combination of prompt engineering and regular expressions can successfully extract the desired source code from the models' output.

4 Enhancing LLM Capabilities

4.1 RAG

Retrieval-Augmented Generation (RAG) [24] is an AI framework designed to enhance relevance and accuracy of LLM outputs by grounding them in external knowledge sources. This allows LLMs to bypass their inherent knowledge boundaries by accessing relevant external documents. Instead of solely relying on the data it was trained on, a RAG system retrieves relevant information from a knowledge base (like documents, databases, or the web) and incorporates it into the prompt given to the LLM.

A standard RAG procedure involves initially indexing and loading a typically vast, domain-specific external knowledge repository. When a user poses a query, a retrieval mechanism pinpoints and extracts relevant documents or knowledge units from this repository. The extracted information is then integrated into the prompt presented to the LLM. By utilizing this augmented context, combined with the initial query, the LLM can produce a more knowledgeable, accurate, and contextually relevant response. Fundamentally, RAG acts as a bridge between the LLM's inherent knowledge from training and external knowledge, which is often more current or specific to a domain, allowing it to generate more reliable and grounded answers. This approach avoids the necessity of frequent fine-tuning, ultimately resulting in more informative and precise responses.

The utilization of RAG for code generation, despite its promise, is still limited in scope. Some noticeable works employed RAG to improve code-related tasks like summarization, generation, and completion. HGNN [10] uses GNNs for code summarization by retrieving similar code. REDCODER [19] retrieves and integrates relevant code snippets for code generation. ReACC [13] leverages both lexical copying and semantic referencing for code completion. DocPrompting [26] uses retrieved code documentation to generate code based on natural language queries. RepoCoder [23] iteratively retrieves and uses code analogies across repository files for better code completion.

Retrieval-augmented code generation has seen advancements, but addressing the challenges of under-represented languages requires significant further exploration.

4.2 Agentic AI

Agentic AI refers to AI systems that can act autonomously to achieve specific goals in complex and dynamic environments with limited direct supervision [22]. These systems, often composed of multiple AI agents, can interact with their environment, reason, make decisions, and take actions with limited human intervention. They can also learn from feedback and adapt their behavior over time [3].

The difference between traditional LLM or RAG-based code generation and agentic AI systems can be illustrated through the task of code generation. In contrast to the multi-stage process involving prompting, code manipulation, compilation, and execution required with LLMs or RAG, an agentic system allows developers to initiate code generation and execution with a single prompt, automating the subsequent steps. The whole process is be executed without human intervention.

Both academia and industry have shown significant interest in the potential of agentic AI to enhance automated code generation. For example, MetaGPT [7] incorporates human workflow into collaborative multi-agent LLM systems. It breaks down complex code-related tasks into specific, actionable procedures. These procedures are then assigned to five different LLM-based agents. The accuracy of the generated tests from MetaGPT is 79% for HumanEval benchmark. AgentCoder [8] employs a multi-agent architecture comprising three specialized agents: a programmer, a test designer, and a test executor. This division of roles results in AgentCoder in more efficient and effective code generation. The performance of AgentCoder on the HumanEval benchmark was groundbreaking, achieving a pass@1 of 96.3%.

Cursor[7] is a novel AI-powered IDE built on VSCode[8], offering a unified AI interface with two primary modes: Ask and Agent. Ask mode allows users to query specific code, understand complex functions, find patterns, and explore their codebase. Agent mode enables AI-driven code changes, refactoring, feature implementation, debugging, and the generation of tests and documentation. To complete tasks, Agent mode employs a structured process. It begins by analyzing the user's request and the codebase context. The agent might then explore the codebase, documentation, and the web to find relevant information. Following this analysis, it breaks down the task and plans the necessary code changes, which it then implements across the codebase. The agent presents the differences for user approval before providing a summary of the modifications. Agent mode primarily uses LLMs like Claude 3.5 Sonnet, with the option to switch to Claude 3.7 Sonnet, Gemini Pro 1.5, or GPT-4o.

5 Code Generation with RAG

To achieve automated translation of 150 C Lingua Franca test files to Python with minimal human oversight, we adopted the Retrieval-Augmented Generation (RAG) approach powered by LlamaIndex[9]. We developed an end-to-end system that automatically translates Lingua Franca code from C to a Python target, given a list of input files. This system goes through sequential stages: Retrieval, Generation, and Code Evaluation, as detailed in Fig. 2. For each file in the list of input files, the system starts by fetching the file source code and initiates the Retrieval stage.

At the Retrieval phase, the embedding model, an artificial neural network, encodes the input Lingua Franca C source code into a dense numerical vector representation (embedding). This embedding encodes the semantic information and

[7] https://www.cursor.com/.
[8] https://code.visualstudio.com/.
[9] https://www.llamaindex.ai/.

contextual relationships of the code in a high-dimensional embedding space, facilitating efficient similarity search for semantically related Lingua Franca Python files within a vector database. The vector database is populated during initialization with embeddings of the LF Python codebase, derived using the same embedding model. OpenAI's text-embedding-ada-002[10] served as our embedding model for this experiment. A similarity search within the vector database then identifies relevant Lingua Franca files, which become the retrieved context. We combine this retrieved context with the input Lingua Franca C source code and a pre-established prompt to form a contextual input for the code LLM, which processes this input to generate the desired code output. The pre-established prompt is:

> **Prompt (1):** *The following code is written in Lingua Franca for target C. Based on this code provide an equivalent Lingua Franca code only for Python target. Provide code without any comment or code fences.*
> *Avoid naming any variable, input, or output with Python reserved words.*
> *The code:*

The generated LF code then undergoes syntax validation by the LF code generator. Syntax errors trigger a recursive call to the Code LLM with the same input for a new code generation attempt. If no syntax errors are found, the LF code generator produces LF Python code. With successful LF Python code generation and execution, the generated source code is incorporated into the codebase, and the vector database is updated to include its representation. Failures in generation or execution also initiate a recursive call to the Code LLM for an alternative code candidate, with a limit on the number of recursive iterations to prevent infinite loops.

Table 2. Performance comparison of Code LLMs for LF code translation on 25 LF C files.

Model	# Parameters	Correct syntax	Run Success	Availability
GPT-4o	200	**15**	**8**	[API]
Claude-3-5-sonnet	175	13	6	[API]
StarCoder2	15	13	7	[Checkpoint]
Qwen2.5-Coder	14	15	7	[Checkpoint]
CodeLlama	13	14	7	[Checkpoint]
DeepSeek-Coder-6.7	6.7	15	7	[Checkpoint]
Codegeex2	6	13	7	[Checkpoint]
DeepSeek-Coder-1.3	1.3	13	6	[Checkpoint]

Following the definition of our system, the subsequent requirement was the selection of appropriate Code LLMs. To address this, we elected to perform a

[10] https://platform.openai.com/docs/models/text-embedding-ada-002.

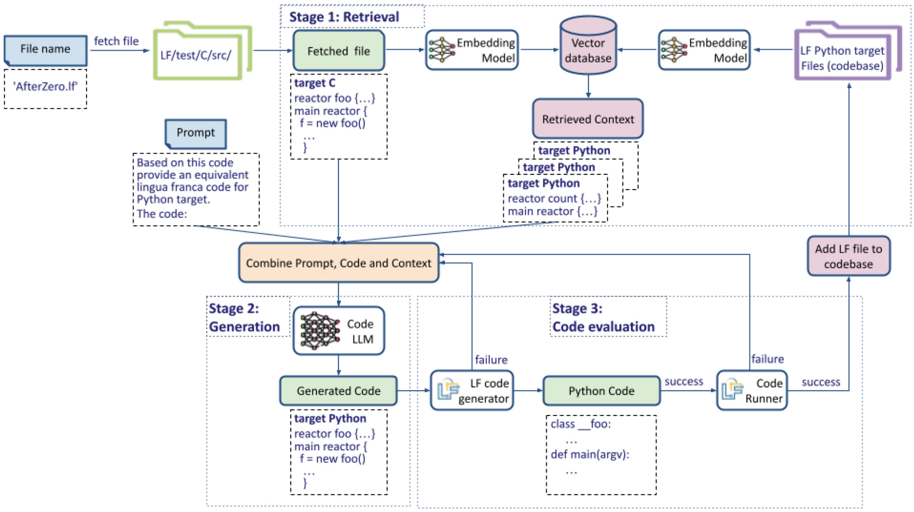


Fig. 2. End-to-end for LF code translation.

comparative analysis of the performance of various Code LLMs in translating a limited set of LF C files (25 files). The costs of APIs and cloud resources constrained us to use a limited number of files, allowing us to include more models in the comparison. We made the choice to incorporate every model that Sect. 3 introduced into our experimental procedure. We report in Table 2 the performance of the selected Code LLMs for LF code translation on the limited set of 25 LF C files. Table 2 provides a comparative overview of the models, listing their size in number of parameters (in billions), the count of syntactically valid generated LF files, the number of generated files exhibiting successful runtime behavior, and their respective availability. Access to proprietary LLMs is facilitated through paid API endpoints, while open-source LLMs are freely distributed on the Hugging Face platform[11], necessitating deployment on local or remote GPU hardware. We conducted this experiment using a remote NVIDIA L4 GPU that had 24 GB of memory. Based on this experimental comparison, GPT-4o exhibited the best results, achieving 15 LF files with correct syntax and 8 successful executions among them. With sizes of 6.7 billion and 14 billion parameters, DeepSeek-Coder-6.7 and Qwen2.5-Coder occupied the second rank in our comparison. The results indicate that Claude-3-5-sonnet, the second largest model (175 billion parameters), ranked lowest, sharing this position with the smallest model, DeepSeek-Coder-1.3 (1.3 billion parameters). DeepSeek-Coder-1.3 exhibited a performance that was only a little less strong than the other LLMs, despite its smaller size. Considering GPT-4o's top performance in our evaluation and the fact that DeepSeek-Coder-1.3 is the sole LLM that our

[11] https://huggingface.co/.

local NVIDIA GeForce RTX 2060 GPU's 6 GB memory can run, we opted to focus our subsequent experiments on these two models.

In Table 3, we present the resuts of the experiment extension to the 150 LF files to translate form target C to target Python. GPT-4o outperforms DeepSeek-Coder-1.3 in code translation by around 5% rate of both syntactically correct code generation and successful execution. This performance differential is significantly smaller than the 31.7% outperformance of GPT-4o (84.9% [17]) over DeepSeek-Coder-1.3 (65.2% [6]) observed on the Python HumanEval benchmark. This observation shows the contribution of the RAG technique in enhancing Code LLMs performance. With a 154 times smaller size, DeepSeek-Coder-1.3 exhibits a performance very similar to GPT-4o. This observation demonstrates the contribution of the RAG technique in improving the efficiency of Code LLMs, as evidenced by DeepSeek-Coder-1.3's ability to achieve near parity with GPT-4o despite a 154-fold difference in size. Here, we don't report the results of LLMs without RAG augmentation, as they were not able to generate any syntactically correct LF files. This observation highlights the importance of RAG in enhancing the performance of Code LLMs for LF code translation tasks.

Table 3. Performance Comparison of Claude and DeepSeek-Coder in Translating 150 LF C Files.

Model	Correct syntax	% Correct syntax	Run success	% Run success
GPT-4o	59	39.33%	29	19.33%
DeepSeek-Coder-1.3	51	34.00%	22	14.67%

With our proposed RAG-based system we were able to correctly generate 19.33% of the desired LF files with GPT-4o LLM. For the remaining files, we need the user to interfere in the correction of the failing generated files. For this purpose, we will utilize the Agentic IDE Cursor.

6 Agentic Assistant

Through manual inspection of the code generated by the proposed system, we uncovered significant issues. Notably, 5% of the generated code files do not include the mandatory '*target Python*' declaration. This omission is critical, as it will inevitably result in syntax errors when processing any subsequent LF code. Furthermore, another prevalent syntactic error involves the inclusion of type declarations for variables, a construct that is incompatible with the Python target. An additional area of non-conformance is the naming of variables with Python keywords. For instance, the system fails to avoid naming input variables 'in'. Input variables often retain their C language name '*in*', which conflicts with Python's syntax and causes errors. We note that the pre-established prompt (see Prompt (1)) instructed the LLM to avoid Python keywords, but the system kept using '*in*' as input variable name. To overcome these issues we utilize IDE Cursor in Agent mode.

To resolve the missing '*target Python*' declaration we provided Cursor's assistant with the subsequent prompt:

Prompt (2): *Identify files without the "target Python" at the beginning and add it.*

The assistant offered to check files for the absence of "target Python" at the beginning and add it if missing. After an initial check, the assistant found that most files already contained "target Python", sometimes with additional settings like timeout. The assistant then asked if it should continue checking all files or focus on a specific set. The latency of this interaction was too slow compared to an IDE's "Find in Files". To improve efficiency, we decided to use a shell script that searches for the files missing the target declaration and appends it. A prompt that might be effective is:

Prompt (3): *Provide a scipt shell that searches for *.lf files, then checks each file if it contains "target Python".*

The assistant provided a shell script designed to find *.lf* files and check if they contain the string "target Python". The assistant included the script itself, an explanation of what the script does, and the commands to make the script executable, and run it. And finally asked *"Would you like me to create this script in your workspace?"*. With a *"Yes"* response, the assistant started an Agent that created the script in the workspace, made the script executable, ran it, and listed the files that don't contain "target Python". Once done, the assistant asked *Would you like me to add "target Python" to these files?*. The assistant iterated through a list of the identified *.lf* files. For each file, the assistant stated its intention to add "target Python" and then confirmed that it had done so. Finally, the assistant provided a summary of the files modified and asked for confirmation to actually apply these changes. After the user's confirmation, the assistant started an agent that added "target Python" to each identified file and finished by providing a summary of the changes made.

To deal with the syntactic error involving the inclusion of type declarations for variables with the Python target, we provided the assistant with a list of target files and a suitable prompt. The list of the files containing type declarations is provided through the assistant context and the prompt instruction is:

Prompt (4): *Remove type declarations.*

For each file in the provided list, the assistant started an agent that checked file content for any type declarations, removed them, then provided a summary of the findings and the updates. The assistant asserted that *the file now has no type declarations, which is consistent with the Python target's requirements.* Finally, it asked to check any other files for type declarations that need to be removed.

When it comes to renaming the input variable '*in*', VS Code's refactoring tool struggled to accurately rename it due to its frequent appearance within other strings. For this purpose, we provided Cursor's assistant with the list of files containing the string *'input in'* and the following prompt:

Prompt (5): *Replace the variable 'in' by 'inp'.*

With this simple prompt, the assistant able to correctly rename most of the occurences of the variable '*in*' avoiding accidental replacements in comments, strings, or other variables with similar names. This process revealed inconsistencies in input variable naming, with a mix of '_*in*' and '*in*_' being used. We reiterated the input variable renaming procedure to build a more consistent codebase. With a cleaner codebase we enhanced RAG capabilities for future code generation.

Pursuing the procedure of code refactoring, we observed numerous occurrences of the *'{:d}'* pattern in the source files. This old-style pattern is used in Python string formatting to format integers, primarily in print statements and error messages. This pattern is discourage in favor of f-strings formatting for better readability, conciseness, and performance. Therefore, we searched for files containing the old-style '%' formatting, added the list of these files to the assistant context, and provided the following prompt:

Prompt (6): *Update old-style string formatting with f-strings formatting.*

Through iteration of the specified files, the assistant successfully implemented the f-string formatting style.

After the series of interactions involving the 150 files generated by our proposed RAG-based system using the GPT-4o LLM, and with all the updates in place, we conducted the code evaluation as outlined in Fig. 2, stage 3. The results reported in Table 4 show improvements in both syntactically correct code generation and successful execution rates. Using Cursor's Agentic Assistant for the generated code correction effectively fixed errors caused by the influence of the source code.

Table 4. Performance Comparison of RAG using Claude and its combination with the Agentic Assistant in Translating 150 LF C Files.

Model	Correct syntax	% Correct syntax	Run success	% Run success
RAG	59	39.33%	29	19.33%
RAG with Agentic Assistant	73	48.67%	52	34.67%

7 Conclusion

Our study demonstrates the effectiveness of combining RAG with agentic AI systems for the task of translating Lingua Franca programs with target language C into comparable programs with target language Python. Through empirical evaluation, we show that the RAG-enhanced pipeline enables smaller open-source models like DeepSeek-Coder-1.3B to perform competitively against larger models such as GPT-4o. This proves that RAG can significantly enhance the performance of code LLMs, allowing them to generate syntactically correct code and execute it successfully, even with limited model sizes. A refinement using the Cursor agentic IDE successfully corrects common syntax and semantic errors, improving the quality of the generated codebase. This hybrid approach not only reduces the reliance on proprietary APIs but also presents a scalable, efficient strategy for code transformation and maintenance in real-world development environments.

The faster translation time achieved by our RAG-based system on 150 files, compared to the interactive IDE assistant, motivate us to pursue full automation of the code correction process. As future work, we propose developing an autonomous agent for code correction to enhance code generation efficiency. However, the limited resources and data for the under-represented Lingua Franca language present a significant challenge in building such an agent.

References

1. Anthropic: The Claude 3 Model Family: Opus, Sonnet, Haiku. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf
2. Cassano, F., et al.: MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. IEEE Trans. Softw. Eng. **49**(7), 3675–3691 (2023). https://doi.org/10.1109/TSE.2023.3267446
3. Chan, A., et al.: Harms from increasingly agentic algorithmic systems. In: Proceedings of the 2023 ACM Conference on Fairness, Accountability, and Transparency, FAccT 2023, pp. 651–666. Association for Computing Machinery, New York (2023). https://doi.org/10.1145/3593013.3594033
4. Chen, M., et al.: Evaluating large language models trained on code. ArXiv **abs/2107.03374** (2021)
5. GLM, T., et al.: ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools (2024)
6. Guo, D., et al.: DeepSeek-coder: when the large language model meets programming - the rise of code intelligence. ArXiv **abs/2401.14196** (2024). https://arxiv.org/abs/2401.14196
7. Hong, S., et al.: MetaGPT: meta programming for multi-agent collaborative framework. arXiv preprint arXiv:2308.00352 (2023). **3**(4), 6
8. Huang, D., Zhang, J.M., Luck, M., Bu, Q., Qing, Y., Cui, H.: Agentcoder: multi-agent-based code generation with iterative testing and optimisation. arXiv preprint arXiv:2312.13010 (2023)
9. Hui, B., et al.: Qwen2.5-Coder Technical Report (2024)
10. Liu, S., Chen, Y., Xie, X., Siow, J., Liu, Y.: Retrieval-augmented generation for code summarization via hybrid GNN. arXiv preprint arXiv:2006.05405 (2020)
11. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a Lingua Franca for deterministic concurrent systems. ACM Trans. Embed. Comput. Syst. **20**(4) (2021). https://doi.org/10.1145/3448128
12. Lozhkov, A., et al.: StarCoder 2 and The Stack v2: the next generation. ArXiv **abs/2402.19173** (2024). https://arxiv.org/pdf/2402.19173

13. Lu, S., Duan, N., Han, H., Guo, D., Hwang, S.w., Svyatkovskiy, A.: ReACC: a retrieval-augmented code completion framework. arXiv preprint arXiv:2203.07722 (2022)
14. Macedo, M., Tian, Y., Cogo, F., Adams, B.: Exploring the impact of the output format on the evaluation of large language models for code translation. In: Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering, FORGE 2024, pp. 57–68. Association for Computing Machinery, New York (2024). https://doi.org/10.1145/3650105.3652301
15. Microsoft: promptbase (2023). https://github.com/microsoft/promptbase. Accessed 22 Apr 2025
16. Murali, V., et al.: AI-assisted code authoring at scale: fine-tuning, deploying, and mixed methods evaluation. Proc. ACM Softw. Eng. **1**(FSE) (2024). https://doi.org/10.1145/3643774
17. OpenAI, et al.: GPT-4 Technical Report (2024). https://arxiv.org/abs/2303.08774
18. Pan, R., et al.: Lost in translation: a study of bugs introduced by large language models while translating code. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE 2024, pp. 1–13. ACM (2024). https://doi.org/10.1145/3597503.3639226
19. Parvez, M.R., Ahmad, W., Chakraborty, S., Ray, B., Chang, K.W.: Retrieval augmented code generation and summarization. In: Moens, M.F., Huang, X., Specia, L., Yih, S.W.t. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2021, pp. 2719–2734. Association for Computational Linguistics, Punta Cana, Dominican Republic (2021)
20. Puri, R., et al.: CodeNet: a large-scale AI for code dataset for learning a diversity of coding tasks. In: Vanschoren, J., Yeung, S. (eds.) Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual (2021). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html
21. Rozi re, B., et al.: Code Llama: open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)
22. Shavit, Y., et al.: Practices for governing agentic AI systems. Research Paper, OpenAI (2023)
23. Zhang, F., et al.: RepoCoder: repository-level code completion through iterative retrieval and generation. arXiv preprint arXiv:2303.12570 (2023)
24. Zhao, P., et al.: Retrieval-augmented generation for AI-generated content: a survey. arXiv preprint arXiv:2402.19473 (2024)
25. Zheng, Q., et al.: CodeGeeX: a pre-trained model for code generation with multilingual evaluations on HumanEval-X. ArXiv **abs/2303.17568** (2023). https://arxiv.org/abs/2303.17568
26. Zhou, S., Alon, U., Xu, F.F., Wang, Z., Jiang, Z., Neubig, G.: DocPrompting: generating code by retrieving the docs. arXiv preprint arXiv: 2207.05987 (2022)

CASP: An Evaluation Dataset for Formal Verification of C Code

Niclas Hertzberg[1(✉)], Merlijn Sevenhuijsen[2,3], Liv Kåreborn[1], and Anna Lokrantz[1]

[1] AI Sweden, Stockholm, Sweden
{niclas.hertzberg,liv.kareborn,anna.lokrantz}@ai.se
[2] Scania, Södertälje, Sweden
merlijn.sevenhuijsen@scania.com
[3] KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. Recent developments in Large Language Models (LLMs) have shown promise in automating code generation, yet the generated programs lack rigorous correctness guarantees. Formal verification can address this shortcoming, but requires expertise and is time-consuming to apply. Currently, there is no dataset of verified C code paired with formal specifications that enables systematic benchmarking in this space. To fill this gap, we present a curated evaluation dataset of C code paired with formal specifications written in ANSI/ISO C Specification Language (ACSL). We develop a multi-stage filtering process to carefully extract 506 pairs of C code and formal specifications from The Stack 1 and The Stack 2. We first identify C files annotated with formal languages. Then, we ensure that the annotated C files formally verify, and employ LLMs to improve non-verifying files. Furthermore, we post-process the remaining files into pairs of C code and ACSL specifications, where each specification-implementation pair is formally verified using Frama-C. To ensure the quality of the pairs, a manual inspection is conducted to confirm the correctness of every pair. The resulting dataset of C-ACSL specification pairs (CASP) provides a foundation for benchmarking and further research on integrating automated code generation with verified correctness.

Keywords: Evaluation Benchmark · Formal Verification · Specification-Implementation Pairs · Dataset Creation

1 Introduction

Large Language Models (LLMs) for code generation have achieved remarkable results in recent years, showing strong performance on tasks such as generating syntactically valid functions and providing code completions [7,8,20,23]. While LLMs demonstrate value across many coding tasks, their utility remains limited in domains with strict safety and quality requirements, such as safety-critical

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 63–82, 2026.
https://doi.org/10.1007/978-3-032-07132-3_5

systems, where software failures can lead to severe consequences. A key limitation is that code generated by LLMs cannot be reliably guaranteed to be correct.

In contrast to the probabilistic nature of LLMs, formal specification languages offer a potential solution to the nondeterministic behavior of LLMs. These formal specification languages provide robust means to specify program behavior, which can then be verified using formal verification tools. However, the adoption of such formal specification languages faces practical challenges: the pairs of formal specifications and associated code must be manually written, which is both time-consuming and requires expertise.

Given that manually creating such pairs is time-consuming, LLMs provide a promising application of being able to generate code from specifications or vice versa.

The task of generating specifications from code and code from specifications is arguably a most fundamental use case for applying LLMs to formal specification generation, directly targeting the most time-consuming tasks. Furthermore, like the related task of correcting non-verifying code, these tasks provide an easily interpretable success metric through formal verification since the generated pairs either verify or fail to verify.

However, evaluating the feasibility and progress of such an approach requires ways to measure the performance of LLMs in the form of dedicated evaluation datasets. We present such a dataset, consisting of verified pairs, that is specifically designed to benchmark this core generative capability.

Existing datasets containing ACSL specifications [1,2,4,11,25] and C code have two significant limitations. First, the datasets are limited in size and therefore lack the breadth needed to cover diverse real-world use cases. Lacking large and diverse datasets, researchers cannot draw general conclusions regarding LLMs' abilities to generate formally verified pairs.

Second, existing datasets are typically distributed as collections of whole C source files, where specifications are embedded as comments. This format requires a non-trivial parsing and extraction step to isolate individual specification-implementation pairs before they can be used in an evaluation pipeline for pair-generation tasks. These two limitations hinder progress in integrating formal methods into the software development process.

Our work addresses this fundamental gap by providing a dataset with sufficient breadth and volume to give researchers a reliable benchmark. Our dataset consists of C code and formal specifications in ANSI/ISO C Specification Language (ACSL), chosen for its adoption by the Frama-C [10,18] verification platform, which is used in both academic research and industrial contexts for verifying critical properties of C programs [12,27]. We call this dataset CASP, short for C-ACSL specification pairs. We create CASP by first sourcing all C files from *The Stack v1* and *The Stack v2* repositories. The C files are then filtered using three steps: (1) identifying and retaining high-quality C files containing formal specification in ACSL; (2) ensuring that the ACSL-annotated C files formally verify, and attempting to correct these if they do not verify (3) extracting individual function implementations and their formal specifications to obtain function-specification pairs.

This paper offers the following contributions:

1. We present CASP: a unique dataset of C code paired with associated formal specifications in ACSL, accessible at: https://huggingface.co/datasets/nicher92/CASP_dataset. The dataset contains 506 pairs. These pairs are systematically extracted from large-scale open-source datasets (The Stack v1 and v2) and are formatted as pairs in order to be amenable to LLM evaluation.
2. We share the complete files from which each pair was taken, accessible at: https://huggingface.co/datasets/nicher92/CASP_source_files. The number of files is much larger than previously available public datasets, offering significantly more data for training and evaluation. Additionally, the files – and by extension the pairs– are all "minimally complete" – meaning the files have no dependencies other than the standard C libraries.
3. We detail our filtering, verification, and post-fixing procedures, ensuring that each file, as well as each code-specification pair, formally verifies and is consistent with one another.

This dataset fills a gap by providing a valuable resource for benchmarking and training LLMs on the task of specification generation from code and vice versa. By offering a conveniently formatted dataset consisting of verified, minimally complete pairs of specifications and associated code, our work supports the development of advanced tools for software verification by contributing to the creation of more reliable software systems.

The rest of this paper is organized as follows. We first provide a brief overview of formal verification, in particular the ANSI/ISO Specification Language (ACSL) in Sect. 2. Then in Sect. 3, we review existing formal specification datasets and their limitations. Section 4 explains our data collection methodology, followed by our file verification process in Sect. 5. We then describe how we divided the files into specification-function pairs Sect. 6. Section 7 presents the composition and key statistics of the dataset, with a discussion and analysis in Sect. 8. Finally, Sect. 9 presents our conclusions and suggests directions for future work.

2 Background

This section provides background on the specification language and verification tools used in our dataset. In particular, we describe the ANSI/ISO C Specification Language (ACSL) and the Frama-C verification framework, with a focus on the WP and RTE plugins used to check correctness and runtime safety.

In the dataset, we focus on ANSI/ISO C Specification Language (ACSL), which enables the formal verification of C code. The language is designed for use with the Frama-C verification framework, a framework for static analysis and deductive verification [3].

2.1 ANSI/ISO C Specification Language

ACSL is a contract-based specification language that allows formal verification of C programs by defining preconditions, postconditions, invariants, and memory access constraints. It is designed to be used with the weakest precondition plugin of Frama-C.

```
1 /*@
2     requires \valid(x) && \valid(y);
3     assigns *x, *y;
4     ensures *x == \old(*y) && *y == \old(*x);
5 */
6 void swap(int* x, int* y) {
7     int temp = *x;
8     *x = *y;
9     *y = temp;
10 }
```

Fig. 1. ACSL specification and associated implementation in C for a function swapping two integers.

Figure 1 demonstrates an ACSL-annotated function that swaps the values of two integer pointers. The `requires` clause (line 2) specifies that both variables `x` and `y` must be valid pointers before execution. The `\valid` predicate ensures that the pointers reference accessible memory. The `assigns` clause (line 3) explicitly states that the function modifies the memory locations pointed to by `x` and `y`, making side effects explicit. The `ensures` clause (line 4) guarantees that after execution, the values of variables `x` and `y` have swapped. The `\old` keyword refers to the values before function execution, ensuring that the function correctly swaps the values.

2.2 The Frama-C Framework

Frama-C is a modular analysis framework for C programs that supports a variety of verification techniques, including runtime error detection and deductive verification. In this work, we use two of its key plugins: WP and RTE.

The `WP (Weakest Precondition)` plugin generates proof obligations called goals from ACSL-annotated C code using weakest precondition calculus. These obligations are passed to SMT solvers (e.g., Alt-Ergo, Z3, CVC4), which attempt to automatically prove that these goals within a given timeout and number of steps.

The `RTE (Runtime Error)` plugin instruments the program with ACSL annotations that check for common runtime errors, including division by zero, null pointer dereference, invalid memory access, and integer overflow. The WP plugin then verifies these additional checks as part of the deductive verification process. Together, WP and RTE enable Frama-C to verify both functional correctness and runtime safety.

3 Related Work

This section reviews prior work relevant to our dataset, including large-scale source code collections, existing datasets containing formal specifications in C, and recent efforts to combine ACSL with automated code generation using LLMs.

3.1 Large-Scale Source Code Datasets

The availability of large source code datasets is fundamental for training and evaluating large language models (LLMs) on code-related tasks. Notably, The Stack v1 [19] and v2 [22], created as part of the BigCode Project[1], provide vast repositories of permissively-licensed source code across numerous programming languages. The Stack v1 comprises approximately 546 million files totaling 6.4 TB, covering 358 programming languages. The subsequent release, The Stack v2, significantly expanding this collection to over 3 billion files (67.5 TB) in more than 600 languages, further enhancing the diversity and volume available for model training and evaluation.

Table 1. Total number of files and number of C files in The Stack v1 and v2. Note that M refers to million, and B refers to billion.

	The Stack v1	The Stack v2
Total number of files	5.46M	3B
Number of C files	19.88M	40.88M

3.2 Formal Specification Datasets Used in Literature

Existing collections of C code annotated with ACSL specifications primarily originate from research projects, serve as educational materials, or have uncertain origins.

Datasets developed in research contexts often function as case studies for formal verification techniques [15,25] or as benchmarks for evaluating analysis tools [4], typically created through manual ACSL annotation of C code. A significant educational resource is the ACSL tutorial [5], designed to teach specification writing through hands-on exercises. This tutorial contains numerous examples, many intentionally left incomplete for learners to finish, reflecting its pedagogical goal.

Common characteristics of these available datasets include small size, formatting as individual files (sometimes with dependencies to other files like .h headers), and a structure tailored to their specific origin or teaching objective rather than forming a larger corpus designed for evaluating LLMs.

[1] https://www.bigcode-project.org/.

Table 2. Existing Formal Specification datasets for C code.

Dataset source	ACSL annotated C files	Minimally complete verified programs
Frama-C-problems [25]	51	9
X509-parser [1]	6	0
Verker [14]	48	1
ACSL By Example [15]	86	3
WP examples [5]	295	134
ACSL proved [13]	34	10
VecoSet [4]	15	14

3.3 Previous Work on LLMs for C Code and ACSL Specifications

Prior research has explored the intersection of C programming and ACSL specifications, particularly in the context of leveraging LLMs for code generation and verification. Minal et al. [24] investigated the feasibility of using LLMs to generate automotive safety-critical embedded C code from both natural language and ACSL specifications. Their study demonstrated the potential of producing compilable and partially verifiable code without iterative backprompting or fine-tuning, though the limited scope of their case studies highlighted the need for more extensive datasets. Similarly, Sevenhuijsen et al. [26] developed a tool that employs a two-step process of initial code generation followed by iterative improvement using feedback from compilers and formal verifiers. The tool successfully generated verified C programs for a majority of the problems in their benchmark set, underscoring the effectiveness of combining formal specifications with automated code generation. However, the relatively small number of code samples in these studies indicates a pressing need for larger, more comprehensive datasets to draw stronger conclusions and enhance model performance.

Similar work has begun to infer ACSL specifications automatically from C code. Granberry et al. prompt GPT-4 with source code plus test inputs and static-analysis warnings, then refine the output of the model until it verifies in Frama-C [16]. Wen et al. apply heuristic post-processing to GPT-4 predictions, correcting syntax and adding safety clauses so the resulting contracts verify more reliably [28]. Together, these studies show that coupling large language models with formal-methods feedback is a promising route to automatic ACSL specification generation.

4 Dataset Collection

Our methodology for collecting and curating a dataset consisting of C functions paired with their corresponding ACSL specifications. We employ a three-step data collection process, shown in Fig. 2. This section focuses on the first step of our dataset creation, where we gather a large collection of source files and then

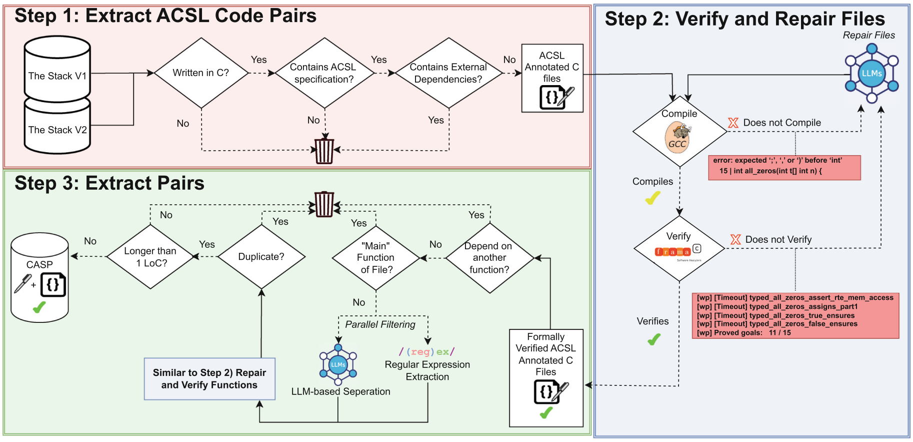


Fig. 2. Overview of the three-step dataset construction pipeline. `Step 1` involves using files from The Stack v1 and v2, which are filtered to identify C files containing ACSL specifications without external dependencies. Step 2 compiles and verifies each annotated file, and files that fail this are automatically repaired using LLMs and re-evaluated. Step 3 transforms successfully verified files into minimal specification-implementation pairs and includes them in CASP.

iteratively filter the files in order to isolate ACSL-annotated C files without other dependencies.

4.1 Downloading the Stack

As an initial step in our data processing pipeline, all files tagged as being written in `C` were downloaded from the deduplicated versions of Stack V1 and V2. Table 1 shows the total number of files in The Stack (V1 and V2) and the subset identified as C files.

4.2 Extracting Files Containing ACSL Specifications

After our initial collection of files, we applied regular expression filters in several steps, in order to extract files that contain ACSL-like annotations. The regular expression patterns were authored by a formal methods expert and are detailed in Appendix A. Each pattern is associated with a *confidence label* indicating whether it can appear in multiple formal languages, such as VeriFast [17] (*possible overlap*), or is unique to a specific language (*exclusive*). We iteratively applied stricter quality filters, only keeping files containing specific ACSL syntax, which left us with 2958 files, as can be seen in Table 3.

Table 3. Dataset collection and filtering process. Starting from The Stack (deduplicated versions 1 and 2), we progressively filter files to identify those containing ACSL specifications. The strict filtering uses the pattern `/\*@.*?(predicate|requires|ensures).*?\*/`.

Processing Step	Count	Description
Initial Data Collection		
Stack 1 (deduplicated)	8,625,559	Raw code samples
Stack 2 (deduplicated)	17,093,668	Raw code samples
Combined total	25,719,227	Total initial code samples
Dataset A Creation		
Initial regex filtering	14,525	Files matching basic patterns
Stricter filtering	5,916	Files with formal specifications
ACSL filtering	2958	Files with ACSL
Standard/No import filtering	1180	Minimally complete files

4.3 Minimally Complete C Files

Many of the collected C files depended on code from other files or non-standard libraries. These dependencies were often complex, making the extraction of verifiable functions and specifications challenging.

To address this issue, we applied an additional filtering step: we only retained files that are self-contained, with dependencies limited to standard C libraries. We refer to these as "minimally complete files" since the specifications and functions they contain can be analyzed independently without requiring external code.

After this stage of our pipeline, we retained 1180 minimally complete C files (see Table 3) that contained ACSL specifications.

5 Verifying and Curating CASP Source Files

This section describes the second step of our three-step process depicted in Fig. 2. Specifically, it describes our method for verifying the correctness of our minimal complete files and our attempts to correct files that do not compile or formally verify.

5.1 Method

To formally verify the minimally complete C files, we attempt to verify if the C implementation meets the formal specifications in these files. The verification process is done by two plugins of Frama-C [9], which we described in Sect. 2.2. We verified each source file using Frama-C version 30.0 (Zinc) with the WP and RTE plugins. The verification was performed using multiple SMT solvers to combine their strengths: Z3 version 4.8.12, Alt-Ergo version 2.6.0, and CVC4 version 1.8. We configured the WP plugin with a 500,000-step limit and a 60-second timeout per proof goal.

For each source file, we either successfully completed all of the goals set by the WP and WP-RTE [6] plugins of Frama-C, or we retrieved the non-verifying goals from Frama-C and attempted to repair the specification. Files that fail to meet the specifications were sent to an LLM (Gemini 2.0 Flash) along with the non-verifying goals from Frama-C and a prompt requesting to update the code such that all goals are verified. We then iteratively attempted to correct each failing file for a maximum of seven iterations. Through this process, we ended up with 469 verified files in our final dataset (Table 4).

Table 4. File Analysis Summary

Category	Count
Minimally complete files	1180
Minimally complete files verified without modifications	292
Minimally complete files verified with modifications	177
Total verified minimally complete Files	469

Prompt Engineering. Our approach to prompt design was iterative, refining the instructions for the LLM based on patterns of verification errors observed in its outputs. We focused particularly on addressing common verification challenges, such as proper contract clause ordering, memory access specifications, and strategic assertion placement to guide proofs. The LLM was prompted to make minimal changes to the code, and also to output what it "thinks" the user intended with their code in order to limit deviation from the original code. The complete prompt used in our processing pipeline can be found in Appendix B.

6 CASP Pair Creation

This section explains step three of the three-step process mentioned in Fig. 2. It describes the means for separating the verifying files into specification implementation pairs.

6.1 Motivation for Specification-Implementation Pairs

Creating verified formal specifications and function pairs offers three advantages over verified C files. First, using specification-function pairs provides a more decoupled method for evaluating LLM performance on formal verification tasks than using entire files. For example, since each specification corresponds to a function implementation, it is possible to assess the generative capabilities of an LLM given a formal specification. Second, these pairs ensure that the specification is logically consistent with the implementation and practically implementable. This addresses a fundamental challenge in formal methods where abstract specifications may contain logical inconsistencies or unrealizable requirements. Third, the structure of the dataset supports bidirectional evaluation—from specification to code and vice versa – which in turn supports a broader range of research questions related to the generative capabilities of LLMs.

6.2 Minimally Complete Files to Minimally Complete Pairs

To create minimally complete pairs from minimally complete files, we selected function implementations according to the following requirements:

- The functions do not depend on other functions in a file.
- The functions are not main functions.

The decision to focus on standalone functions was guided by two primary factors. Methodologically, it creates a constrained test that directly evaluates an LLM's core ability to translate between a specification and an implementation. Practically, the task of identifying, extracting, and verifying the complete context for functions with numerous dependencies from large codebases is often intractable. Our approach therefore ensures that each pair in CASP is a self-contained and verifiable unit.

We utilized two parallel pipelines in order to extract pairs of ACSL specification and C implementation from our source files: One based on utilizing an LLM and one based on regular expressions. Each minimally complete file that fulfilled the above requirements was sent to both pipelines.

The regular expression-based pipeline comprises three steps: first, extracting function implementations, second, their corresponding ACSL specification if present, and third, any additional dependencies needed for verification of the pair. Any functions without associated specifications were removed.

The LLM-based pipeline (Using Gemini 2.0 Flash) consisted of prompting the model to extract functions, their associated specification, and any additional dependencies[2]. Regular expression for function extraction and prompt can be found in Appendix C.

We take the union from the resulting pairs from both pipelines, which were then verified by Frama-C, similarly as described in Sect. 5.1; any remaining unverified pairs were manually post-fixed.

The resulting union of pairs, 513 in total, from the two pipelines was then filtered in two steps: We performed exact deduplication of the C implementations and only kept function implementations longer than one line of code, leaving us with 506 pairs.

[2] With additional dependencies we mean standard imports, type and variable declarations, logic predicates, etc.

7 Dataset Statistics

In this section, we provide a statistical overview of our dataset. First, in Fig. 3 we show the length distribution measured as lines of C code for our pairs, and find that most of the CASP pairs are short to medium in length. Correspondingly, the number of lines of ACSL for each pair can be seen in the distribution plot in Fig. 4, most specifications are short to medium in length, with some more complex outliers.

Beyond characterizing the CASP pairs themselves, we also analyze the diversity and novelty of the verified C source files from which these pairs were derived. Comparing the source files to existing file-based ACSL datasets allows us to assess the breadth of our data collection. Since our dataset is collected from open-sourced code, there is a possibility of overlap between CASP and other open-sourced datasets containing ACSL. In order to measure potential overlap, we embed our files and compare the semantic similarity of our files to existing datasets. The comparison is done in two ways: first, using a t-SNE plot in two dimensions, and second, by a nearest neighbour comparison. An in-depth analysis of the semantic distributions can be found in Sect. 8.2.

7.1 Semantic Distribution of File Contents

We downloaded existing datasets (see Table 2) containing ACSL specifications and C code. We filtered each C file in all datasets – including the CASP source files – so that all files were minimally complete, ensuring a fair comparison between datasets. We then embedded the files that verify using CodexEmbed [21] – a model specifically developed for code retrieval. We used the 2B parameter model variant. The embedding model has a maximum context length of 4096, which means that longer code samples were truncated, potentially affecting their representation.

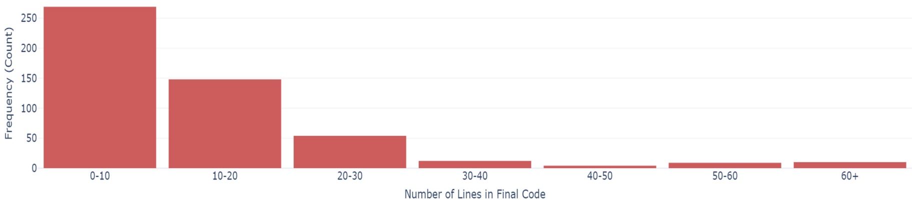


Fig. 3. Distribution of lines of C code for each CASP pair, excluding specification and imports. Most programs are short to medium in length. The X axis indicates lines of code, and the Y axis indicates a number of occurrences. Outliers over 60 total lines of code are binned together.

Using these high-dimensional embeddings, we visualized the semantic relationships between files using t-SNE (Fig. 5) and quantitatively analyzed the similarity distribution by calculating nearest neighbor distances (Fig. 6).

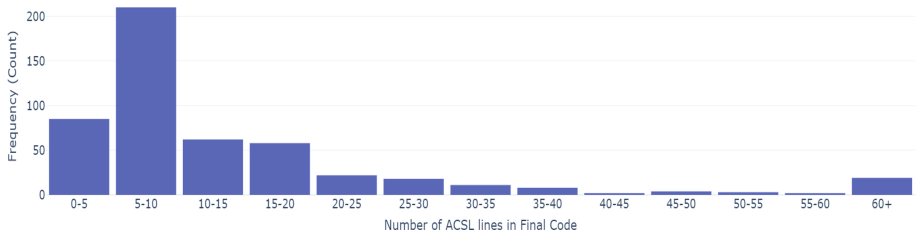


Fig. 4. Distribution of total lines of ACSL for each CASP pair. The X axis indicates the total number of lines of ACSL, and the Y axis indicates the number of occurrences. Outliers over 60 total lines of ACSL are binned together.

8 Discussion and Analysis

In this section, we describe our findings on the current state of openly sourced, ACSL-annotated code. We then provide an analysis of CASP and the source files from which CASP was derived. Finally, we discuss our method for LLM-based specification repair along with implications for verification and specification generation.

8.1 Current State of Affairs and the Need for CASP

Our investigation confirmed a significant challenge for researchers in automated verification and specification generation: the pronounced scarcity of openly accessible C code accompanied by ACSL annotations. Furthermore, where such code exists, a substantial portion exhibits quality issues, often failing verification by Frama-C. This scarcity presents challenges for researchers seeking to build comprehensive datasets for LLM training or benchmarking purposes. We hypothesize that the main reason for this scarcity is that while ACSL is a fairly standard verification language, much of the code where it is present is not openly available on GitHub with permissive licensing, and is therefore not included in The Stack 1 and 2.

Despite this scarcity, CASP is the largest openly released dataset containing ACSL specifications and C code so far. Additionally, the dataset is formally verified and formatted in a way conducive to evaluating LLMs. Another key strength of CASP lies in its inherent diversity. Since the collected code samples were authored by numerous different programmers, they exhibit considerable variety in implementation styles, algorithmic approaches, and specification patterns. This diversity strengthens the utility of our dataset for various research applications, as it represents a broad spectrum of real-world specification practices rather than the more uniform patterns that might emerge from a single team or project.

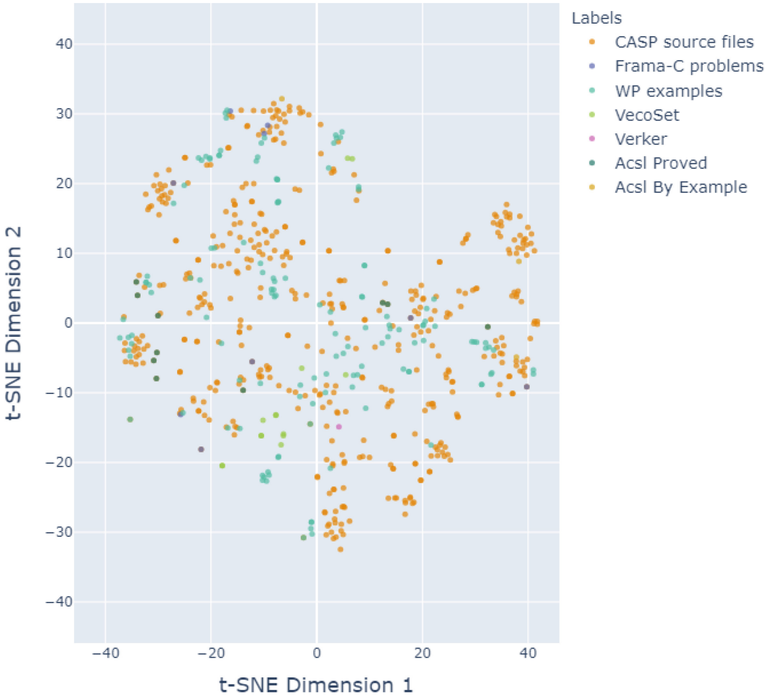


Fig. 5. t-SNE visualization of embeddings from various datasets that verifies without any external imports (CASP source files, Frama-C problems, VecoSet, etc.). The plot shows the projection of high-dimensional embeddings into a 2D space, where proximity suggests similarity. Colors indicate the source dataset as shown in the legend. Any difference in embeddings indicates a difference in file content.

8.2 Dataset Composition and Analysis

The embedded source files from CASP – when visualized using t-SNE in Fig. 5 – occupy a broader range of the semantic space and therefore show greater diversity than existing datasets, encompassing most regions where samples from existing datasets are located. We hypothesize that there are two reasons for this: The CASP source files are substantially more numerous than existing datasets, and the source files originate from multiple sources and numerous different authors. It should be noted that the ACSL annotated C files from other datasets often contain imports from .h files – which we do not include – causing many of the files to not verify.

Furthermore, our analysis (see Fig. 6) reveals that several files from the datasets Frama-C Problems, ACSL Proved, WP Examples, and CASP are similar to at least one other file in one of the datasets. Beyond these clusters of similarity, we found a broad distribution of datapoint relationships across the similarity spectrum. Overall, we find that there is fairly limited overlap between CASP and previous datasets, since only approximately 35 CASP source files are

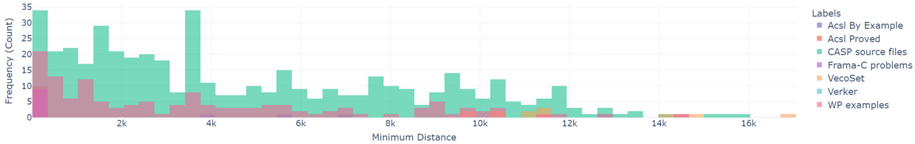


Fig. 6. Nearest neighbours of various datasets (CASP source files, Frama-C problems, VecoSet, etc.). The plot shows the distance between each embedded source file, where the source file is from and how similar its nearest neighbour is. A dataset containing a wide variety of files would contain more points to the right on the x axis and vice versa.

very similar to some other file in any of the other datasets. One explanation for this limited overlap is that we remove files that are dependent on files not found in the standard libraries, including .h files. Many of the C files in the existing datasets contain .h files and are, therefore, naturally filtered if found.

8.3 LLM-Based Specification Repair and Pair Extraction

While not the primary focus of this work, our approach to repairing faulty ACSL-annotated C files using LLMs showed meaningful success.

Our methodology successfully corrected 177 files out of the 888 that initially failed verification, representing a 19.9% success rate among files requiring modification. The success rate highlights the challenging nature of formal specification repair, even for advanced LLMs. Nevertheless, the fact that nearly one-fifth of problematic specifications could be automatically corrected suggests potential for improvement in this area. One explanation for this rate of success is related to the limited amount of ACSL-annotated C code that is openly available: it is difficult to train an LLM to understand the syntax and semantics of a formal language that the LLM has barely encountered previously. Another reason could be that the non-repairable files themselves were poorly written and therefore difficult for an LLM to repair, especially given the constraint not to deviate from the original implementation.

In addition to repairing ACSL-annotated C files, we also utilized LLMs for function extraction. We found that LLMs struggle in some cases but can be complemented by straightforward rule-based methods based on regular expressions. This combined approach proved to be more successful than using either approach on its own.

9 Conclusions and Future Work

This paper has demonstrated our approach to creating CASP: a dataset consisting of 506 verified and deduplicated C code functions paired with ACSL specifications.

CASP has only been publicly available since June 2025, it serves as a timely and uncontaminated benchmark for all current models and any future models with a training data cutoff before this date, but future work should also address the long-term maintenance of CASP as a benchmark. To mitigate the risk of data contamination from future model training, a portion of the dataset could be reserved for a private test set, and our methodology could be used to generate new versions of the benchmark over time.

Additionally, our findings suggest several promising directions for extending this work. We suggest two main avenues of exploration, extending CASP and specific dataset applications.

A natural extension could involve exploring different data sources to expand our data set. For example, academic papers and technical documentation often contain high-quality specifications created by experts that could yield additional examples of formally verified pairs. Synthetic data generation, using CASP for seed prompts with modifications to promote greater diversity in the generated code samples, is another promising avenue to explore.

As CASP was created first and foremost with evaluation of LLMs in mind, a natural next step in terms of applications would be to evaluate a wide variety of different LLMs on generating code from formal specifications and vice versa. Future work, possibly following an extension of the dataset, might also explore training specialized models for formal verification tasks or developing automated tools for specification generation and repair.

A Regex Patterns Used

See Table 5.

B Gemini Prompt

The following is the prompt used to instruct Gemini to correct ACSL specifications in C code based on Frama-C error messages.

Listing 1.1. Main prompt given to Gemini

```
Your task is to correct ACSL specifications based on C code and error
    messages from Frama-C.
Your goal is to repair the ACSL specifications so that they pass Frama-C's
    verification.
Do not alter the C code unless absolutely necessary.
Focus on correcting ACSL specifications to address common errors such as:

Invalid ACSL syntax
Type mismatches in ACSL expressions
Loop invariants that are not strong enough or incorrect
Precondition or postcondition failures
Memory access errors or incomplete memory specifications
Incomplete or incorrect assigns clauses
Timeout issues in proof obligations

Pay special attention to:

Using precise memory specifications: \valid, \valid_read, \separated as
    appropriate
Ensuring loop invariants are strong enough to prove postconditions
```

Table 5. Patterns for ACSL, and Verifast Annotations

ACSL
/@(?:(?!@/)[\s\S])*?\brequires\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\bensures\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\bassigns\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\binvariant\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\baxiomatic\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\blemma\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\bpredicate\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\blogic\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\bbehavior\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\bdisjoint behaviors\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\bcomplete behaviors\b(?:(?!@/)[\s\S])*?/
/@(?:(?!@/)[\s\S])*?\bassumes\b(?:(?!@/)[\s\S])*?/
//@\s*\brequires\b
//@\s*\bensures\b
//@\s*\bassigns\b
//@\s*\binvariant\b
//@\s*\baxiom\b
//@\s*\blemma\b
//@\s*\bassert\b
loop invariant
loop assigns
loop variant
\\old\b
\\at\b
\\nothing\b
\\max\b
\\min\b
\\result\b
\\forall\b
\\exists\b
\\sum\b
\\sizeof\b
\\valid\b
\\valid_read\b
\\is_finite\b
Verifast
/@(?:(?!@/)[\s\S])*?\bopen\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\brequires\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\bensures\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\bassert\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\bfold\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\bunfold\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\blemma\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\bpredicate\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\bopen\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\bclose\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\binvariant\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\bpointer\(\b(?:(?!@/)[\s\S])*?@/
/@(?:(?!@/)[\s\S])*?\bmalloc_block(?:(?!@/)[\s\S])*?@/
//@\s*\binclude\b
//@\s*\brequires\b
//@\s*\bensures\b
//@\s*\bassert\b
//@\s*\bfold\b
//@\s*\bunfold\b
//@\s*\binvariant\b
//@\s*\blemma\b
//@\s*\bopen\b
//@\s*\bclose\b
//@\s*\bleak\b

```
Adding explicit loop assigns clauses to clarify what loops modify
Using complete behaviors and disjoint behaviors when appropriate
Adding strategic assertions to guide the prover
Using \exists and \forall quantifiers correctly
Ensuring that array bounds are properly specified
Not to add undefined variables that are not defined in the code previously.
Loop assigns is not allowed after loop variant so they need to be prior to
     the loop variant
Wrong order of clause in contract: behavior should be before complete or
     disjoint for example
Using correct syntax for behaviors: each behavior should be declared
     separately using the behavior keyword, not enclosed in braces; complete
     behaviors and disjoint behaviors should be followed by a comma-separated
     list of behavior names without braces
The ACSL specifications for a function should be above the function, not
     below it.
Avoid adding main() functions if not present in the original code.
In general the changes should not attempt to alter the purpose of the
     original code.

For timeout issues, consider:

Simplifying complex specifications
Breaking down properties into smaller, more provable assertions
Using different specification styles (direct ensures vs. behaviors)
Adding intermediate assertions to guide the proof

Output the corrected file in JSON format, including a brief explanation of
     the changes made and any assumptions.

Input:
C Code (Previous Attempt):
{file_content}
Frama-C Error Message (From Previous Attempt):
{error_message}
```

Listing 1.2. Context prompt for subsequent iterations

```
Context from Previous Gemini Attempt (that produced the code above):
Previous Explanation: {prev_explanation}
Previous Assumptions: {prev_assumptions}
Previous Strategies Suggested: {prev_strategies}
Based on the previous attempt's code, the resulting error message, and the
     previous explanation/assumptions, please refine your corrections to
     address the remaining issues. Focus specifically on the errors
     highlighted in the Frama-C message.
```

Listing 1.3. Context prompt for first iteration

```
This is the first attempt to fix the provided code and error message in this
     refinement cycle. Please analyze the code and error carefully.
```

Listing 1.4. Required output format

```
Output Format:
Please provide your response in the following JSON format:
{
  "explanation": "Explanation of changes made in this attempt",
  "assumptions": "Any assumptions made during this correction process",
  "fixed_code": "Complete corrected code with fixed ACSL specifications here
      ",
  "strategies": "Suggestions for prover strategies if timeout issues persist
      (e.g., specific provers, timeouts, steps)"
}
```

C Function Pair Extraction

Listing 1.5. Regex for extracting non main functions

```
r'''
(?P<signature>
    (?:[a-zA-Z_][\w\s\*\(\),]*?)      # return type and qualifiers
    \s+                               # whitespace
    (?!(main)\s*\()                   # not 'main'
    [a-zA-Z_]\w*                      # function name
    \s*\([^;]*?\)                     # parameter list
)
\s*\{                                 # opening brace of body
'''
```

Listing 1.6. Gemini prompt for function and specification pair extraction

```
You are given a C source file that contains one or several functions with
     corresponding ACSL specifications and additional dependencies.

Your task is to extract all functions that are independent of other functions
     , except `main`, which should be excluded. A function is considered
     independent if it does not call or rely on other user-defined functions
     in the file.

For each such function:
1. Extract the full function implementation (signature and body).
2. Extract the ACSL specification that precedes it (typically marked by /*@
     or //@).
3. Identify and include only the minimal dependencies required for Frama-C
     verification of the function. This may include:
   - `#include` directives (e.g., `<stdbool.h>`)
   - `#define` macros
   - global constants or variables used in the function

Return a JSON object for each function with the following fields:
- "function_implementation": the code of the function (not `main`)
- "acsl": the ACSL specification
- "dependencies": the minimal required includes/defines/globals

If a function depends on another user-defined function in the same file, skip
      it.

C source code:
{file_content}
```

References

1. ANSSI-FR/x509-parser, December 2024. Original-date: 2019-06-03T12:22:56Z
2. Alur, R., Fisman, D., Padhi, S., Singh, R., Udupa, A.: SyGuS-Comp 2018: results and analysis, April 2019. arXiv:1904.07146 [cs]
3. Baudin, P., et al.: ANSI/ISO C specification language version 1.21 (2024)
4. Beyer, D.: Progress on software verification: SV-COMP 2022 (2022). https://doi.org/10.1007/978-3-030-99527-0_20
5. Blanchard, A.: Frama-C and WP tutorial code examples (2025). https://github.com/AllanBlanchard/tutoriel_wp

6. Blanchard, A., Bobot, F., Baudin, P., Correnson, L.: Formally verifying that a program does what it should: the WP plug-in. In: Guide to Software Verification with Frama-C: Core Components, Usages, and Applications, pp. 187–261. Springer (2024)
7. Brown, T., et al.: Language models are few-shot learners. In: Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020)
8. Chen, M., et al.: Evaluating large language models trained on code, July 2021. arXiv:2107.03374 [cs]
9. Correnson, L., et al.: Frama-C user manual for Frama-C 30.0 (zinc) (2024)
10. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
11. Dillig, I., Dillig, T., McMillan, K., Li, B.: Inductive Invariant Generation via Abductive Inference
12. Dordowsky, F.: An experimental Study using ACSL and Frama-C to formulate and verify Low-Level Requirements from a DO-178C compliant Avionics Project. Electron. Proc. Theor. Comput. Sci. **187**, 28–41 (2015). arXiv:1508.03894 [cs]
13. Efremov, D.: evdenis/acsl-proved, March 2016. Original-date: 2017-02-14T16:30:05Z
14. Efremov, D., Mandrykin, M., Khoroshilov, A.: Deductive verification of unmodified Linux kernel library functions. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11245, pp. 216–234. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_15
15. Gerlach, J.: ACSL by Example (Version 22.0.0) for Frama-C 22.0 (Titanium). Fraunhofer FOKUS (2020). https://github.com/fraunhoferfokus/acsl-by-example
16. Granberry, G., Ahrendt, W., Johansson, M.: Specify what? Enhancing neural specification synthesis by symbolic methods. In: Kosmatov, N., Kovács, L. (eds.) Integrated Formal Methods, pp. 307–325. Springer, Cham (2025)
17. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
18. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Form. Asp. Comput. **27**(3), 573–609 (2015)
19. Kocetkov, D., et al.: The stack: 3 TB of permissively licensed source code (2022)
20. Li, Y., et al.: Competition-level code generation with AlphaCode. Science **378**(6624), 1092–1097 (2022)
21. Liu, Y., et al.: CodeXEmbed: a generalist embedding model family for multilingual and multi-task code retrieval. *arXiv preprint* arXiv:2411.12644 (2024)
22. Lozhkov, A., et al.: Starcoder 2 and the stack V2: The next generation (2024)
23. Nijkamp, E., et al.: CodeGen: an open large language model for code with multi-turn program synthesis, February 2023. arXiv:2203.13474 [cs]
24. Patil, M.S., Ung, G., Nyberg, M.: Towards specification-driven LLM-Based Generation of embedded automotive software. In: Steffen, B. (ed.) Bridging the Gap Between AI and Reality, pp. 125–144. Springer, Cham (2025)
25. Patnaik, M.: manavpatnaik/frama-c-problems, October 2024. Original-date: 2020-10-27T11:44:14Z

26. Sevenhuijsen, M., Etemadi, K., Nyberg, M.: VeCoGen: automating generation of formally verified C code with large language models. In: 2025 IEEE/ACM 13th International Conference on Formal Methods in Software Engineering (FormaliSE), Los Alamitos, CA, USA, April 2025, pp. 101–112. IEEE Computer Society (2025)
27. Ung, G., Amilon, J., Gurov, D., Lidström, C., Nyberg, M., Palmskog, K.: Post-Hoc formal verification of automotive software with informal requirements: an experience report. In: 2024 IEEE 32nd International Requirements Engineering Conference (RE), pp. 287–298 (2024). ISSN 2332-6441
28. Wen, C., et al.: Enchanting program specification synthesis by large language models using static analysis and program verification. In: International Conference on Computer Aided Verification, pp. 302–328. Springer (2024)

A Voice-Enabled Query Framework for Systems Engineering Artefacts

Lennart Landt[1(✉)], Martin Leucker[1], and Carsten Burchardt[2]

[1] Universität zu Lübeck, Lübeck, Germany
{lennart.landt,leucker}@isp.uni-luebeck.de
[2] Siemens DI SW, Hannover, Germany
carsten.burchardt@siemens.com

Abstract. Model-Based Systems Engineering (MBSE) is increasingly adopted to manage the growing complexity of modern systems, offering a structured, collaborative design approach through modeling languages like SysML. However, its adoption remains challenging due to steep learning curves and the need for interdisciplinary coordination. This paper investigates the use of voice-enabled AI avatars to simplify MBSE interaction by allowing users to query system artifacts using natural language. By integrating a speech-based interface with AI assistants and human-like avatars representing various engineering roles, we aim to lower entry barriers, foster collaboration, and support diverse perspectives inherent in MBSE. Our proposed pipeline converts model data into a machine-readable format for large language models (LLMs) to generate contextualized, role-based responses. We explore the potential of using MBSE models as a knowledge base for AI and examine how such tools can enhance system model comprehension. Contributions include a prototype implementation, support for multi-turn interactions, and an initial evaluation of the approach.

1 Introduction

Model-Based Systems Engineering (MBSE) is gaining increasing attention worldwide due to several compelling reasons [18]. The complexity of modern systems continues to rise, necessitating more sophisticated approaches to system design and management. The use of modeling languages such as SysML allows for multiple perspectives on a system, facilitating interdisciplinary collaboration and providing a single source of truth [12].

However, applying MBSE in practice presents several challenges. MBSE requires expert knowledge and intensive training. There is a significant (time) investment necessary to become a fully MBSE-enabled developer [9]. The complexity of projects also comes with a multidisciplinary team, which in itself takes up resources for the necessary collaboration efforts [9].

To simplify the employment of MBSE, we explore in this paper whether voice-enabled AI avatars can be used to query artifacts to help users become acquainted with new projects or learn about project artifacts that are not their

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 83–102, 2026.
https://doi.org/10.1007/978-3-032-07132-3_6

main concern. The objective of this approach is to create a human-like interaction with the system, characterized by low thresholds and the use of natural language. The speech-based interface lowers the barrier to entry for users unfamiliar with technical modeling tools, enabling intuitive access through natural language. This fosters inclusivity and supports collaborative engineering scenarios where verbal communication is the norm. The artificial intelligence is considered to function as a "third party in the room". To this end, we employ AI assistant systems to formulate queries that integrate user input with a role-based system introduction and the information gathered from the MBSE model. This AI assistant back-end is integrated with an AI avatar front-end, wherein human-like 3D avatars convey the system's response to the user and function as an immediate interlocutor. The utilization of diverse avatars to impersonate distinct MBSE roles results in the generation of varied responses to the same inquiry and data. This will contribute to the objective of achieving human-like interactions, wherein posing a single question to a group of individuals results in a range of responses that reflect each person's areas of expertise and experiences. This mirrors the MBSE philosophy, where system understanding emerges from the interplay of multiple stakeholder perspectives—such as systems engineers, software architects, and domain experts—each represented by a distinct avatar.

Using AI for our problem may be advantageous because MBSE models are highly structured, rigidly standardized, and based on formal languages like SysML [12]. This standardization allows AI tools to be trained on consistent patterns. Related work has shown that AI can be used in various ways in conjunction with MBSE. For example, AI can be used as assistance in the modeling process [11], support the extraction, classification, and linking of stakeholder requirements to model elements [21], and offers applications when MBSE is linked with Digital Twins [1].

Our high-level solution involves using a pipeline where the model serves as a knowledge base and context provider. We introduce a method for efficiently converting model data into a machine-readable format for transmission to a LLM. The user's question is combined with a system prompt and this model data to generate a response. Our research questions include:

- How can MBSE models be used as a knowledge base for AI assistants?
- Could AI be useful in increasing the understandability of system models?

Our contributions include enabling a speech-based interaction for the users, providing answers based on different roles within the engineering project, allowing complex sequences of interactions and queries to be performed, describing the implementation, and conducting a first, limited evaluation.

2 The Modeling of Systems

In this section we will give a short overview of the topics Systems Engineering (SE) and Model-Based Systems Engineering (MBSE) in order to show the methodology behind our approach. In addition, this section aims to illustrate the rationale behind our decisions, providing a clear understanding of the factors that influenced our approach.

2.1 System Engineering

> "Systems Engineering is a transdisciplinary and integrative approach to enable the successful realization, use, and retirement of engineered systems, using systems principles and concepts, and scientific, technological, and management methods." [17]

The paradigm of Systems Engineering encompasses a goal-oriented, holistic approach to problem-solving. It considers the entire product lifecycle from concept development through to system implementation. A central concept in Systems Engineering is "systems thinking," where a system is viewed as an artifact that is hierarchically fragmented into components. These components all contribute to a single goal that none of them could achieve on their own. The product lifecycle in Systems Engineering spans from concept development through system implementation to the retirement of the system. [22]

The methodology of Systems Engineering aids in the development and handling of the rising complexity of multidisciplinary systems. Its goal is to achieve an interdisciplinary optimum within a predefined time and budget framework. Systems Engineering connects and structures the different disciplines using models. The three main parts of Systems Engineering are tailoring, methods, and modelling. Tailoring refers to the customization of Systems Engineering, methods are proven problem-solving procedures, and modelling helps manage the complexity of systems by developing and examining representations that span the entire system lifecycle [12].

The ISO/IEC norm 15288 defines and subdivides the system life cycle in 25 processes, that are assigned to one of four main processes: The agreement processes, the organizational project-enabling processes, the technical management processes and the technical processes. Each of the processes can be described through an input, process activities and an output, as shown in the example of the business and project analysis process in Fig. 1.

[12] describes a role model that provides a complete overview of all the different tasks and responsibilities in Systems Engineering. It serves as a reference for assigning roles in an organization. Fifteen different roles are described, each with its own responsibilities, tasks, and expertise. Those roles are: System Architect, SE Process Manager, Requirements Manager, Modeling Engineer, Project Lead, Configurations Manager, Information Manager, Entrepreneur, System Security Manager, Implementation Manager, V&V Engineer, Life Cycle Manager, Interface Manager, Technical Manager, Stakeholder Interaction Manager. For example, the Technical Manager is responsible for technical development decisions on all system levels. His tasks are the systematic preparation of decisions, the holistic decision-making, and the resolving of conflicts and consensus building. His professional and social skills lie in a comprehensive understanding of the system, leadership skills and negotiating abilities.

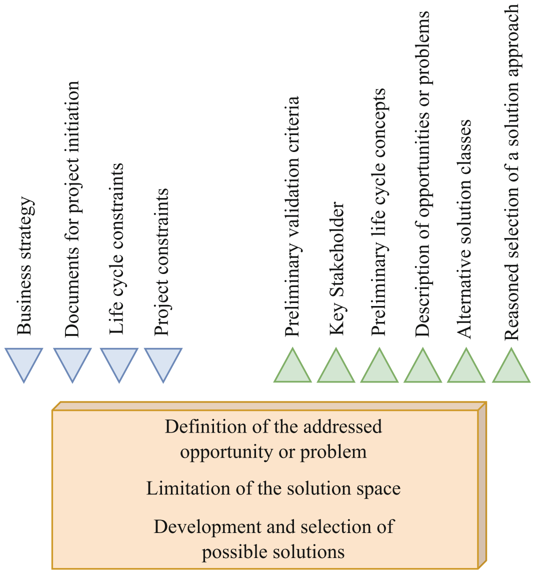


Fig. 1. This figure shows exemplary one of the SE-processes: the business and project analysis process. Each process has strictly defined inputs, activities, and outputs. For example, one input here is the business strategy, one activity the limitation of the solution space and one output the key stakeholders. Based on [12].

2.2 Model-Based Systems Engineering

> "MBSE is the formalized application of modelling to support system engineering processes [...]" [13]

One shortcoming of known SE methods is working with document-centered presentation forms of development results and intermediate statuses. These forms are not linked, redundant, and therefore inconsistent. By adding model-based methods to SE, it evolved into MBSE, which provides a "single source of truth" [22].

MBSE itself is defined by the INCOSE, the International Council on Systems Engineering [16], as "a formalized application of modelling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later lifecycle phases".

In the realm of Model-Based Systems Engineering (MBSE), the MBSE-triangle consisting of tool, language, and methodology forms the foundation of a robust approach. In our case, we have chosen Capella as the tool, SysML as the language, and ARCADIA as the methodology. This combination provides a comprehensive framework for our MBSE activities.

There are several mainstream MBSE methodologies, e.g. Magic Grid [10], ARCADIA [3], HarmonySE [15] [13]. Every one has its own procedure to model a system, its own advantages and disadvantages. In our approach, we use ARCADIA as the methodology due to its well-structured and systematic procedure. ARCADIA is excellent for mapping different systems perspectives to AI-avatars, providing a clear and organized framework for analysis and design. Furthermore, its integration with Capella made it a prime candidate for our approach. The structured nature of ARCADIA ensures that all aspects of the system are thoroughly considered and modelled.

The ARCADIA-method (ARChitecture And Design Integrated Approach) [4] consists of a four-part system modelling process. Those four parts are: The Operational Analysis, the System Analysis, the Logical Architecture and the Physical Architecture.

- Operational Analysis (OA): "What system users must achieve." [4]
- System Analysis (SA): "What the system must achieve for the users."' [4]
- Logical Architecture (LA): "How the system will work to meet expectations." [4]
- Physical Architecture (PA): "How the system will be built." [4]

The OA typically starts with the identification of the users of the system and the relationship between each other and the system. Also, the information, that gets used in each activity and interaction, should be captured here. Furthermore, it is a key point in the OA to identify the stakeholders and their activities. With having the users and stakeholders in mind, one defines the operational capabilities, the operational processes and scenarios, and the operational modes and states. [6]

The focus of the SA is to define how the system can satisfy the needs captured in the OA. In order to do this, the SA creates an external functional analysis. For example, here the system boundaries and external interfaces get captured and the functions of system and actors get identified. Also, the capabilities of each system actor are identified. For this, it is a good praxis to create functional chains and build behavioral scenarios. [8]

While the system during the System Analysis is viewed as a black box, in the LA the system gets defined on a deeper level. Logical Components are formalized in this step. Although the LA defines the core of the system, the Logical Components still remain abstract structural elements. The main intend of the LA is to build an abstraction of the system, that is just detailed enough for making decisions without getting lost in small details. Those will be later defined in the Physical Architecture. [5]

The objective of the PA is to define the concrete components of the system. Here the final architecture with all the functions gets defined, and the behavioral components get deployed. [7]

In addition, there is also a great variety of different modelling tools in the landscape of MBSE. Examples include open-source tools like Capella [2] or proprietary software like Rhapsody [14] and MagicDraw [23]. Again, we do not go into detail about the differences between those tools. We chose Capella for our approach mainly because of its open-source nature. Capella is a powerful tool that supports the ARCADIA methodology and offers extensive features for systems modeling. Its open-source nature allows for customization and extension, making it an ideal choice for our project. We aim to implement an Add-On for Capella based on our approach in the future. However, the underlying methodology of our approach could be adapted to other tools, demonstrating the flexibility of our strategy.

SysML (System Modeling Language) is the quasi-standard modelling language in the domain of MBSE. The language was first released in 2007. It is based on the object-oriented modeling language UML (Unified Modeling Language), that is widespread in the field of software engineering. SysML adapted UML from a software-only perspective to a complete system view by integrating mechanical and cyber-physical systems. Where UML uses classes, SysML incorporates blocks as structural elements to model every type of entity in the system. The nine different types of SysML-diagrams get clustered in three groups: Requirement-diagrams, structural diagrams and behavior-diagrams. [12] SysML provides a standardized way to represent systems, facilitating communication and collaboration among stakeholders. Its widespread adoption ensures compatibility with various tools and methodologies, making it a versatile choice for our MBSE activities.

3 AI, Voice Recognition, LLMs

The solution that has been devised for this purpose involves the incorporation of artificial intelligence (AI), with a particular emphasis on large language models (LLMs). LLMs demonstrate considerable potential for processing natural language, recognizing patterns, and analyzing voluminous data sets. Despite their advantages, there are still considerations to be made when using LLMs. For instance, the models have a tendency to "hallucinate", which refers to the generation of erroneous information. It is imperative to construct a robust guard rail system around the LLM to enhance its accuracy and reliability. Technologies such as Retrieval Augmented Generation (RAG) and vector databases are employed to enhance the performance of LLMs by integrating the capacity to generate text and retrieve pertinent information from extensive databases. The present study focuses on the utilization of OpenAI and ChatGPT-4o, as OpenAI is a leader in the development of advanced AI models and ChatGPT-4o is one of the most powerful language models currently available. The decision to utilize OpenAI and ChatGPT-4o was guided by their demonstrated capacity to deliver high precision and versatility in executing complex linguistic tasks. These models have been thoroughly documented and have garnered significant support from the developer community, thereby facilitating their integration into our solution. Notably, the Realtime API and the Assistant API offered by OpenAI appear to hold considerable promise for the proposed solution.

3.1 Comparison Realtime API and Assistant API

To handle queries, for our approach two solutions are viable. On the one hand, the OpenAI Realtime API provides a voice-to-voice dialogue system, that computes User-input and AI-outputs with low latency [20]. An alternative is to use the OpenAI Assistants-API, that offers a simple way of creating an assistant, although Speech-to-Text and Text-to-Speech need to be handled externally [20].

Table 1 gives an overview of the differences between those two APIs. The Realtime API has an advantage in the handling of speech-input to speech-output, with an integrated solution for voice activity detection, which results in a more dynamic conversation and lower latencies. It lacks a good solution for handling longer conversations, without losing historic information, and a direct tool support. In this work, we decided to use the Assistant API due to its better handling of conversation history and a more flexible setup for testing different Voice-to-Text & Text-to-Voice tools, different AI-models and different ways of combining the user input with the gathered knowledge from the SE-model.

Table 1. Comparison between the OpenAI Realtime API and the Assistants API.

Feature	Realtime API	Assistants API
Conversation History	conversations	threads
System Prompt	Manually through messages-array	Directly via instructions
Input + Knowledge	Indirectly by manually adding to the message-array	Directly by combining first and then pushing as *thread_message*
Voice Integration	Integrated with *voice activity detection* (VAD)	Separate integration
Tool Support	Not directly	Yes (Code, Retrieval, etc.)
Streaming	Yes (easy and direct)	Limited
Latency	Low	High
Flexibility	High, but more effort required	Very high with lower effort
Additional Notes	–	Tighter integration to the vector store
	–	Will be replaced with *Responses API* in 2026

3.2 Configuring Assistant API

The configuration of the Assistant API is a crucial part of our approach. A key point here is to implement a methodology on how to instruct the different AI assistants through their individual system prompt. Some parts of the system prompt can be applied to all assistants. Instructions like "You are a helpful, but also critical assistant in the field of Model-Based System Engineering." or "Answer in a speech-like way." are applied to all assistants. But it is important to have a distinction between the assistants, since one key part of MBSE is the aspect of having multiple different views on the system. In this manner, our approach utilizes multiple unique AI assistants to incorporate this concept in our solution. Furthermore, it is essential to have the AI sticking close to the input, without hallucinating missing information or misinterpreting data. MBSE provides us with a solid foundation for such a methodology, but even in the domain of MBSE there are different standards one could use. For our approach, we found three viable methodologies for characterizing AI assistants. First, one could take the defined SE processes as systems prompt. They have defined input, activities, and outputs and are easy to understand. Furthermore, they can easily be adapted to a broad applicability across domains. On the downside, the processes are not strictly standardized, when it comes to the details. In addition, they have a limited semantic depth for interpretation with AI, since interpretations and terms vary across different organizations.

An approach closer to human roles offer the SE roles adopted in [12]. Due to their resemblance to real project roles, they are easy to adapt to AI assistants. This could translate in a human-like interaction with the AI. One could use the list of responsibilities, tasks, and expertise as a system prompt for the assistants. The biggest issue with this approach is the lack of formalization and standardization. Thus, role definitions can vary significantly depending on the organization and the project. Additionally, this offers a very limited machine interpretability.

For our solution, we decided to use the ARCADIA method as a methodology for our assistant configuration. Despite its drawbacks of high complexity and a challenging mapping to AI avatars, the advantages outweigh the disadvantages. The ARCADIA method offers a formalized and standardized methodology with a strong integration in MBSE tool chains and a high adaptability to different kinds of systems. In particular, the usage of a modeling language like SysML promises a high machine readability.

The comparison between these different approaches can be viewed, summarized, in Table 2. An example of a system prompt, that is tailored to a part of the ARCADIA method, can be seen in 5.

4 Our Solution

In this section, we present our approach. The section is split into two parts: First, the back-end of our solution is explained. The back-end converts the information stored in the system model to a machine-readable format and handles user input. This user input is then combined with the extracted model knowledge to form a single query. Here we show our information and interaction pipeline. Second, the front-end is stated and how the user interacts with the system model through

Table 2. Comparison of MBSE Approaches for AI assistant integration.

Approach	Advantages	Disadvantages
SE Processes	– Clearly defined inputs, activities, and outputs – Easy to understand for human stakeholders – Broad applicability across domains	– Often informal and not strictly standardized – Limited semantic depth for AI interpretation – Varying interpretations across organizations
SE Roles	– Close alignment with human project roles – Facilitates role-based AI assistant design	– Lack of formalization and standardization – Role definitions vary significantly – Limited machine interpretability
ARCADIA Method	– Formalized and standardized modeling – Strong integration in MBSE tool chains – High machine readability (e.g., via SysML-like structures)	– High complexity for human users – Challenging mapping to AI avatars

AI-Avatars. The front-end is responsible for bringing the audio-output from the AI assistant onto the AI avatar, which serves as the human-like interface to the system.

4.1 The Information and Interaction Pipeline

The back-end of our solution is clustered into three different parts: The model-knowledge database pipeline, the AI-assistant pipeline and the AI-avatar pipeline. The connections and processes of these pipelines, which work together to build the whole query framework, are outlined in Fig. 2.

The model-knowledge database pipeline is responsible for the conversion of information contained within system models into a machine-readable format. In this initial approach to employing MBSE diagrams as a data repository, our solution initiates with the exportation of the diagram as an XML file. Due to the considerable size of XML files, even for diagrams of a modest scale, we opted to convert them into a JSON. This approach offers several advantages. Firstly, JSON files typically have a more compact structure, using strictly defined key-value pairs, whereas XML tends to be more nested and includes redundant tags. As a result, JSON files are often significantly smaller in both token count and file size, which improves processing efficiency. While we assume that JSON may offer better fault tolerance and a more structured design for AI interpretation, these claims are based on practical observations rather than empirical evidence. The actual machine readability of a format can depend heavily on the training data and architecture of the AI model. Therefore, we present this approach as a practical choice rather than a definitive superiority claim. The conversion is executed by means of a Python script. It is imperative that each type of MBSE diagram intended for integration into the knowledge base is accompanied by its designated JSON scheme and a Python conversion script. However, the fundamental structure remains unaltered. An example for a such a JSON-scheme can be found in Fig. 5.

In the AI-assistant pipeline, the JSON-files are added to the user input. In our speech-based solution, the human input is transcribed to text and also added to the query. In Algorithm 1 the pseudocode of such a query is shown. First, an AI-assistant is created with its specific system instructions and tools. This is a one-time call per assistant. Then the needed JSON-files get uploaded and their file-id saved. For a new conversation, a thread is created, and its ID gets saved as well for continuing conversations later. The user message is composed of the thread-ID, the user-input, that got transcribed, and the file-IDs of the JSON-files. This message gets sent to the thread and a run is started. The textual AI-answer is then converted from text to speech and send to the AI avatar pipeline. Here, the audio file gets sent to the client script that handles the handover to the frontend. Via a gRPC (gRPC Remote Procedure Call) request, an open-source framework for communication between distributed systems using HTTP/2, the audio file gets pushed to the *Audio2Face Streaming Audio Player* of the front-end.

Algorithm 1: Assistant Interaction with JSON File

Input: User input, JSON file
Output: Assistant response

Create assistant with instructions and tools;
Upload JSON file and store `file_id`;
Create new thread and store `thread_id`;
send_user_message_with_json(`thread_id`, `user_input`, `file_id`) Send message to thread;
  return *response*
Set `user_input` to query string;
Call `send_user_message_with_json` and store response;
Start run with `assistant_id` and `thread_id`;
return *run*

4.2 Interacting with the Model

For the interaction with the model, our solution explores the possibilities of using 3D-AI-Avatars as a direct dialogue partner for speech-based low threshold interaction with the system. A foundational principle of (MB-)SE is the recognition of the system from multiple perspectives. The proposed solution endeavors to incorporate this through the utilization of AI avatar technology.

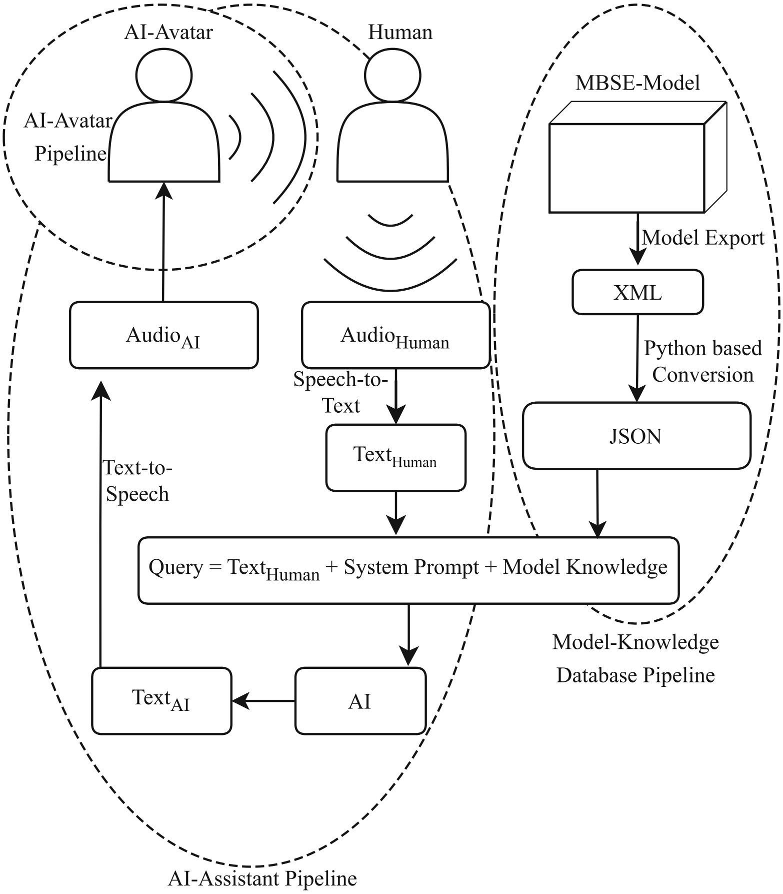


Fig. 2. The figure shows the whole query framework. The *AI-Avatar Pipeline* is responsible for the front-end, where the output audio from the AI assistant is translated to the 3D avatar. The *AI-Assistant Pipeline* handles the human input, converts the speech to text, forms the query and sends the assistant's output to the avatar pipeline. In the *Model-Knowledge Database Pipeline* the model diagrams are exported, converted to JSON and send to the query.

The objective is to reduce the complexity of the system and to lower the barrier for using systems engineering. This objective would be realized if the interaction with the system model were to emulate a human interaction. The utilization of 3D AI avatars presents a compelling approach for facilitating human-machine interaction. Typically, when posing a question to an AI system, the response received is a single, definitive answer. It is possible to obtain multiple responses to a single inquiry through the utilization of our approach. This phenomenon may be compared to posing a question to a group of individuals, resulting in a variety of responses based on the expertise of each person in the group. Speech-based AI avatars facilitate seamless integration into work and thought processes, thereby promoting efficiency and effectiveness in professional interactions. The solution under discussion has been designed to address the specific challenges posed by brainstorming-like situations that frequently arise during the development process. The avatars presented here are intended to function as collaborative partners in the brainstorming process, with each avatar representing a distinct expertise and system knowledge.

For the front-end, the NVIDIA Omniverse™ Audio2Face [19] is used. The tool offers a platform for creating, managing and adjusting 3D human like avatars. It allows the AI-based generation of facial animation and lip synchronization driven only by an audio source. The tool automatically analyzes the given audio-file and animates fitting emotions on the avatar. An example of how such an avatar looks like, can be seen in Fig. 3.

Fig. 3. Here is one of the human-like avatars shown. The image shows one of the examples of the NVIDIA Omniverse Audio2Face tool.

5 Evaluation

In this section, an example system is examined through the lens of our approach to demonstrate and evaluate the solution. The example system facilitates the delineation of the precise mechanisms within our proposed pipelines and enables a more thorough examination.

For illustrative purposes, this paper utilizes a system that was originally developed at the Technische Universität Graz (TU Graz). This system is designed to facilitate rescue operations in mountainous terrain using unmanned aerial vehicles (UAVs), more commonly referred to as *drones*. Here, the focus is on a subsystem: The Rescue Object Detection System is a technological innovation designed to facilitate the identification of objects in rescue operations. The objective of this subsystem is to implement an artificial intelligence-assisted object (person) detection on the live feed from the drone's camera. The scenario unfolds as follows: The software engineer requires information regarding the established software interfaces of the drone's communication system.

To illustrate the solution in a simplified way, two basic roles will be used: The logical architect and the physical architect. The roles will be embodied by two distinct AI assistants, each with its own unique model knowledge, system introduction, and AI avatar. In the following, we will examine the configuration process for the logical architect in greater detail. It should be noted that the configuration process for the physical architect is highly analogous to that of the logical architect.

For the logical architect, we will use the Logical Architecture Diagram (LAB) from the ARCADIA method as the basis for our knowledge base. The LAB constitutes a pivotal element in the Logical Architecture (LA)-Step of the Arcadia method. The LAB is responsible for the allocation of logical functions to the relevant components. The allocation of functions to components enables the functions to be viewed in context. A system model may comprise multiple LAB diagrams, each concentrating on a distinct aspect of the system under consideration. In this particular instance, as illustrated in Fig. 4, the emphasis is directed towards the drone's camera system, with a focus on the interconnections among its components within the Rescue Object Detection System. In the diagram, you can track the signal-chain from the *Control Drone* function in the *Remote Control* component through the *Control Signal* to the *Receive/Send Signal* function in the *Communication System* component. From here, the *Camera Signal* exchange goes to the detection chain in the *Camera System*. If an object is detected in the detection chain, the *Send Notice* function triggers the *Object Detected* exchange to the *Receive/Send Detection Signal* function back in the *Communication System*. This function then has the *Object Detected* exchange with the *Notify User* function in the *Remote Control*. The LAB diagram is given to the input of the Model-Knowledge Database Pipeline. The Python-script then parses the XML-file for the parts, that are defined in the diagram-specific JSON-Scheme. The JSON-Scheme for a LAB diagram can be seen in Fig. 5. Subsequently, the completed JSON file is transmitted to the AI Assistant Pipeline.

When an engineer engages with the front-end, the spoken words are transcribed into text. Subsequent to each human input, a novel query is constructed by integrating the text input of the engineer with the avatar-specific system prompt and the JSON file.

For example, when an engineer asks the system "How reliable is the transmission of the control command to the drone?", the query looks like this:

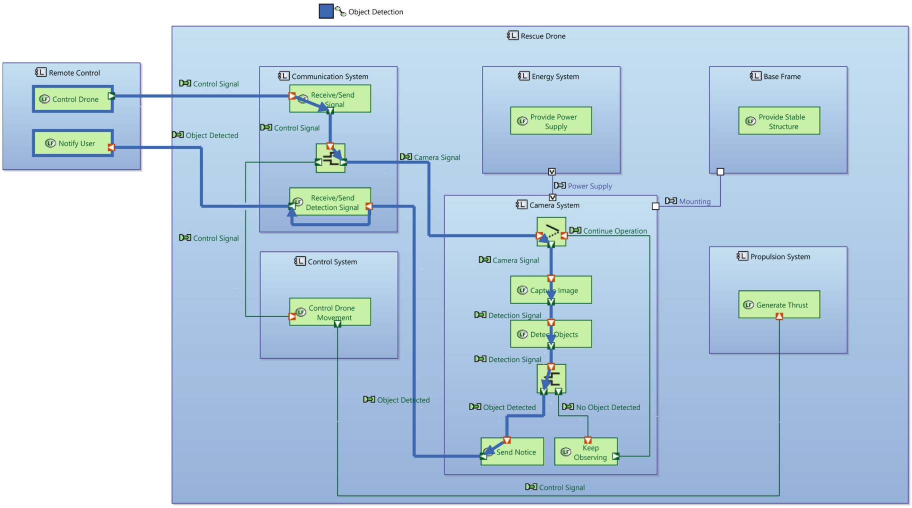


Fig. 4. This is the used Logical Architecture Diagram (LAB). It shows the allocation of logical functions to the component and the exchanges between them.

System Instructions for Logical Architect

"You are a helpful, but also critical assistant in the field of Model-Based System Engineering. You assist in the Logical Architecture following the Arcadia Method. Your input is a user question combined with a Logical Architecture Blank Diagram that allocates the logical functions to the relevant components. Use file_search to answer questions based on the attached JSON, that represents the Logical Architecture Blank Diagram. It contains information about the logical functions, the logical components, the functional allocation and the component exchanges. If a function is allocated to a component, take the function into consideration when answering questions about the component. Answer in a speech-like way."

Example Input

"How reliable is the transmission of the control command to the drone?"

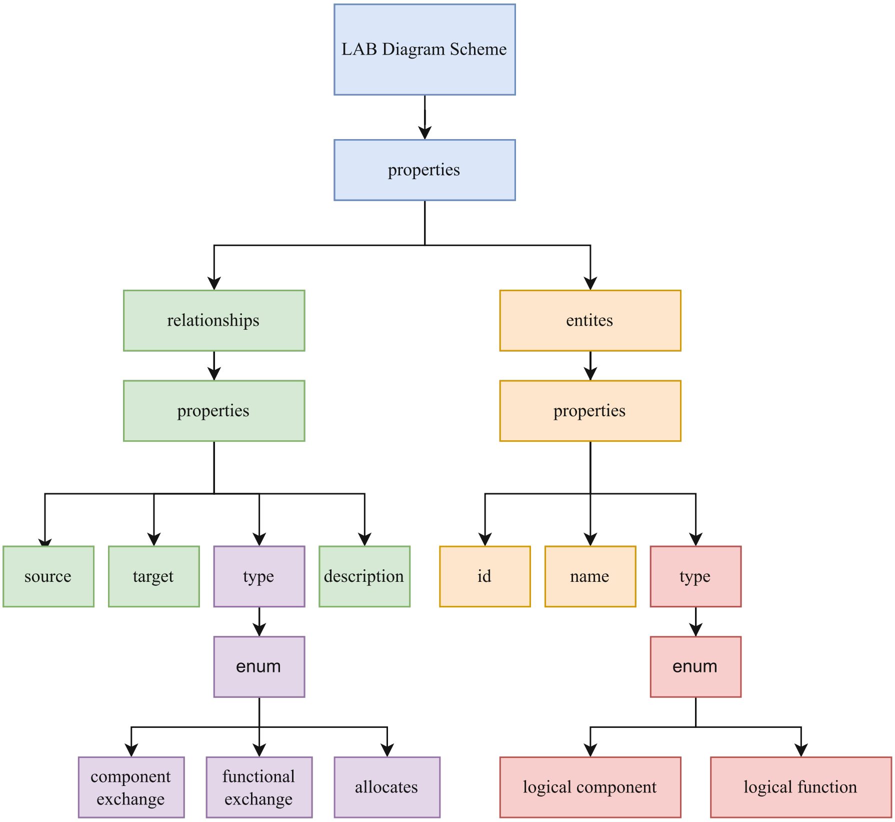


Fig. 5. JSON-scheme used to convert the Arcadia-diagram to a JSON-file. In this example of a LAB-diagram, an object can be a relationship or an entity. A relationship has the tags: source, target, type and description, while an entity has: id, name and type. A relationship can either be a component exchange, a functional exchange or an allocation. An entity can be a logical component or a logical function.

Assistant Output for Logical Architect

"The reliability of the transmission of control commands to the drone is fundamentally reliant on the effectiveness of two main components: the "Remote Control" and the "Communication System. "The "Control Drone" function, responsible for issuing control commands, operates through the "Remote Control" component. The control commands are then transmitted to the drone via the "Receive/Send Signal" function, which is a responsibility of the "Communication System." The setup indicates a structured and clear mechanism for command transmission, which is essential for reliability. However, the actual reliability also depends on several factors beyond the logical architecture, such as the quality of the signal and transmission technology used in the communication system, potential environmental interference, and the robustness of both hardware and software managing these functions. These components must be well-designed and tested under various conditions to ensure that commands are transmitted accurately and promptly, enhancing the reliability of the system."

Assistant Output for Physical Architect

"The transmission of control commands to the drone is done via a 2.4 GHz radio frequency link, utilizing a Radio Transceiver Module on the controller side and a Drone Receiver Unit on the drone. This setup indicates a standard and well-proven approach in remote control systems, and the 2.4 GHz band is widely used for its balance between range and bandwidth. Typically, this setup should be quite reliable assuming there is no significant interference and both transmitter and receiver are within range. However, the reliability can be affected by environmental factors like physical obstacles, misalignment of antennas, or other sources of RF interference, which are common in urban environments. If those factors are controlled, the system should operate effectively."

In this example, the user asks the system about the reliability of the transmission of the control command in natural language in spoken words. To answer this question, one AI assistant takes the role of a logical architect and gets the converted LAB diagram as a knowledge base. The other assistant poses as physical architect and gets a Physical Architecture Blank Diagram (PAB). The assistants answer in regard to their assigned roles. The logical architect focuses on the functional logic of the system. The AI successfully extruded the necessary information from the diagram. Furthermore, even though the components of interest are not directly linked, the AI builds a relationship between them, based on the functional exchanges between the functions of the components. This is an important point, as it is an indication that our approach is able to recognize the complex relationships in models. Meanwhile, the physical architect describes the real technical solution used in the system and provides details like the communication frequency. The output is provided to the user in the form of spoken words through the avatars. The speech is highly human-like and the real-time translation of the speech into the avatars' facial expressions helps convey human-like interaction.

In future works, a deeper evaluation of the solution will be a primary focus. We propose examining robustness, which includes how well the model data is captured, the accuracy of the model, and its fault tolerance. Additionally, we will look at complexity, specifically how high the complexity of queries can be while maintaining good performance. Finally, we will assess usability, focusing on how the solution integrates into existing workflows and gathering feedback from potential users regarding the comprehensibility and usefulness of the solution. These aspects will enable a comprehensive assessment of the solution and help identify its strengths and weaknesses. By analyzing these criteria, we can ensure that the solution is not only technically sound but also practically applicable and user-friendly. The results of this evaluation will provide valuable insights into the performance and potential applications of the solution, serving as a foundation for further improvements and developments.

6 Discussion

The solution presented in the work offers an integration of system models in an AI-based workflow. The objective of the solution is to provide a human-like interaction with the data captured in the model. The goal is to reduce the complexity of systems by using AI to translate the model to a more comprehensible human-machine interaction.

Section 5 showed the solution in a simplified example. An engineer could ask an AI-avatar, presented in 3D-human-like-avatar as front-end, a question about the system. By transforming a model into a machine-readable format, it could be added to the knowledge pool of the AI assistant, coupled with a specific system instruction, developed from the ARCADIA method. The result is an answer by the AI assistant that took the data gathered from the model in consideration and providing the engineer with system insights.

Although the approach yield first successful results, there are some remarks, that need to be discussed and considered in future works: First, a use-case analysis should be executed to develop and test in which scenarios a voice-based interaction is more suitable than a purely text-based interaction. In addition, we propose a hybrid-model capable to differentiate which part of the answer should be spoken and which part is more convenient to give as a textual answer in addition to the spoken answer.

The evaluation with the example system showed, that our approach works good on simple models and questions that are easily answerable using just one diagram as input. Here a deeper evaluation is needed for complex multi-diagram scenarios.

Furthermore, one has to discuss the converting of the pure model into the JSON-format. On the one hand, besides machine-readability, this conversion prepares our approach to be updated to SysML v2, which API's underlying interchange format is JSON [11]. On the other hand, one has to keep in mind that a conversion brings a conjunction to the original model. It is of the essence to prevent that the development team falls back to changing the JSON-file rather than the original model. Especially for small, quick changes, this could be tempting. For this reason, we propose to have a system in place, that automatically updates the JSON-based knowledge base whenever there is a change to the model and not giving the users the option to manually adapt the JSON-files. We will implement such solution in an upcoming iteration of our approach.

In conclusion, the findings of this work provide valuable insights and lay the groundwork for future research and practical applications in the field.

7 Conclusion

In this work, we presented a novel approach that uses MBSE-models as a knowledge base and 3D AI avatars as a front-end, with the goal of utilizing the data within MBSE models in a human-computer interaction that lowers the threshold for using and understanding MBSE-models. Due to the ever-increasing complexity of systems, it is important to find ways to keep humans in the loop and assist them in comprehending the system. The evaluation showed promising results in enabling a human-like interaction with the system. The engineer was able to ask questions about the system, and the AI-assistant used data gathered from an MBSE artifact to answer them. However, there are still several development tasks and scientific questions that remain open.

To address these, in the next steps, we plan to build a retriever system that utilizes vectorization to integrate the entire MBSE model into a single semantic network. This will enhance the coherence and connectivity of the model data. Additionally, we aim to implement the proposed pipeline using the Realtime-API, focusing on developing a system that efficiently manages the communication thread for faster response times, while maintaining context information and ensuring query efficiency in terms of token usage.

Furthermore, we intend to implement our solution, or at least parts of it, as an add-on for Capella. By establishing a direct link to the modeling software, the Model-Knowledge Database Pipeline will become more efficient and enable new features. We will also evaluate the proposed approach using objective metrics, collaborating with industry partners such as Siemens to test and implement our solution in real-world scenarios.

Lastly, the upcoming of SysML v2 is designed to better resemble natural language, which will significantly enhance the integration of the language with large language models (LLMs). This advancement will further improve the usability and effectiveness of our solution.

References

1. Barricelli, B.R., Casiraghi, E., Fogli, D.: A survey on digital twin: definitions, characteristics, applications, and design implications. IEEE Access **7**, 167653–167671 (2019)
2. Capella: MBSE capella (2025), https://mbse-capella.org/, Accessed 27 May 2025
3. Capella MBSE: arcadia method. https://mbse-capella.org/arcadia.html (nd), Accessed 27 May 2025
4. Castro, H.: MBSE arcadia method: ontology elements and traceability. LinkedIn Pulse (2023), https://www.linkedin.com/pulse/mbse-arcadia-method-ontology-elements-traceability-helder-castro/?trk=article-ssr-frontend-pulse_little-text-block, Accessed 28 May 2025
5. Castro, H.: MBSE arcadia method step-by-step: Logical architecture. LinkedIn Pulse (2023), https://www.linkedin.com/pulse/mbse-arcadia-method-step-by-step-logical-architecture-helder-castro, Accessed 28 May 2025
6. Castro, H.: MBSE arcadia method step-by-step: Operational analysis. LinkedIn Pulse (2023), https://www.linkedin.com/pulse/mbse-arcadia-method-step-by-step-helder-castro/?trackingId=P53d%2FO7w9kmURcPB5XxMOQ%3D%3D&trk=article-ssr-frontend-pulse_little-text-block, Accessed 28 May 2025
7. Castro, H.: MBSE arcadia method step-by-step: physical architecture. LinkedIn Pulse (2023), https://www.linkedin.com/pulse/mbse-arcadia-method-step-by-step-physical-helder-castro?trk=article-ssr-frontend-pulse_little-text-block, Accessed 28 May 2025

8. Castro, H.: MBSE arcadia method step-by-step: system analysis. LinkedIn Pulse (2023), https://www.linkedin.com/pulse/mbse-arcadia-method-step-by-step-system-analysis-helder-castro/?trk=article-ssr-frontend-pulse_little-text-block, Accessed 28 May 2025
9. Chami, M., Bruel, J.M.: A survey on MBSE adoption challenges. In: INCOSE EMEA Sector Systems Engineering Conference (INCOSE EMEASEC 2018), pp. 1–16 (2018)
10. Dassault Syst mes: Magicgrid book of knowledge. https://discover.3ds.com/magicgrid-book-of-knowledge (nd), Accessed 27 May 2025
11. DeHart, J.K.: Leveraging large language models for direct interaction with sysml v2. In: INCOSE International Symposium, vol. 34, pp. 2168–2185. Wiley Online Library (2024)
12. Gräßler, I., Oleff, C.: Systems engineering. Springer (2022)
13. Guo, Y., Zhao, H., Qi, K., Guo, J.: Analysis of the mainstream MBSE methodologies from the modeling practice view. In: Proceedings of CECNet 2022, pp. 146–154. IOS Press (2022)
14. IBM: IBM engineering systems design rhapsody (2025), https://www.ibm.com/de-de/products/systems-design-rhapsody, Accessed 27 May 2025
15. IBM corporation: model-based systems engineering with rational rhapsody and rational harmony for systems engineering – deskbook 3.1.2. https://www.ibm.com/support/pages/model-based-systems-engineering-rational-rhapsody-and-rational-harmony-systems-engineering-deskbook-312 (2010), Accessed 27 May 2025
16. INCOSE: MBSE initiative. https://www.incose.org/communities/working-groups-initiatives/mbse-initiative (nd), Accessed 27 May 2025
17. INCOSE: systems engineering definition. https://www.incose.org/about-systems-engineering/system-and-se-definitions/systems-engineering-definition (nd), Accessed 27 May 2025
18. Lawrence, J.-M., Hossain, N.U.I., Rinaudo, C.H., Buchanan, R.K., Jaradat, R.: An approach to improve hurricane disaster logistics using system dynamics and information systems. In: Madni, A.M., Boehm, B., Erwin, D., Moghaddam, M., Sievers, M., Wheaton, M. (eds.) Recent Trends and Advances in Model Based Systems Engineering, pp. 699–712. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-82083-1_59
19. NVIDIA: Audio2face overview (2023), https://docs.omniverse.nvidia.com/audio2face/latest/overview.html, Accessed 28 May 2025
20. OpenAI: introducing the realtime api (2024), https://openai.com/index/introducing-the-realtime-api, Accessed 28 May 2025
21. Schleifer, S., Lungu, A., Kruse, B., Putten, S., Goetz, S., Wartzack, S.: Automatic derivation of use case diagrams from interrelated natural language requirements. Proc. Des. Soc. **4**, 2725–2734 (2024)
22. Sinnwell, C.: Methode zur produktionssystemkonzipierung auf basis früher produktinformationen. Produktionstechnische berichte aus de FBK, Hrsg, Aurich, J **2** (2020)
23. Systèmes, D.: Magicdraw (2025), https://www.3ds.com/products/catia/no-magic/magicdraw, Accessed 27 May 2025

Integrating LLMs with QC-OpenDRIVE: Ensuring Normative Correctness in Autonomous Driving Scenarios

Julian Müller(✉), Thies de Graaff, and Eike Möhlmann

German Aerospace Center, Institute of Systems Engineering for Future Mobility, Oldenburg, Germany
{julian.mueller,thies.degraaff,eike.moehlmann}@dlr.de

Abstract. This paper investigates on the integration of Large Language Models (LLMs) with the `QC-OpenDRIVE` framework in order to generate syntactically and semantically correct OpenDRIVE files. OpenDRIVE files play an important role in the scenario-based validation of autonomous driving systems as they define the static part (e.g. road layout) on which the function are validated. While LLMs excel at generating code or similar tasks which mostly needs to be syntactically correct, the validation of semantic, especially normative, correctness remains challenging. To ensure norm-adherent correctness of generated OpenDRIVE files this paper proposes an integration of a feedback-loop with LLMs and `QC-OpenDRIVE`. While LLM allow to easily generate different road layouts, they often show issues like missing or unconnected roads or improper continuity. To address this issue, we have implemented E.5.9.1 to ensure geometric continuity between connected roads, which is a key contribution of this paper.

State-of-the-art models are evaluated on three tasks to create OpenDRIVE road networks and validate the results featuring the feedback-loop. Results show that models leveraging Retrieval Augmented Generation (RAG) or internal reasoning and using the feedback loop can generate syntactically and semantically valid outputs after iterative corrections. However, challenges remain to prompt complex scenarios and tasks, especially following geometric rules without explicit feedback. The results demonstrate the necessity of domain-specific normative validation frameworks to prepare the use of LLMs for safety-critical applications. They can be used to enable scalable generation of edge-case scenarios while ensuring compliance with industry standards. This work bridges the gap between automated scenario generation and rigorous validation of reliable autonomous driving systems.

Keywords: Large Language Models · OpenDRIVE · QC-OpenDRIVE · Scenario Generation · Semantic Validation · Autonomous Driving

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 103–116, 2026.
https://doi.org/10.1007/978-3-032-07132-3_7

1 Motivation

Beside designing and implementing autonomous driving systems, testing and guaranteeing the safety of such a system is a necessary and non-trivial task. Wachenfeld and Winner [26] state that a pilot for the German Autobahn would need 6,62 bn. driving kilometers to be twice as good as a human driver with 50% certainty. This way of obtaining a safety prove for every new vehicle on the Autobahn is unrealistic and also economically not feasible. A promising approach is virtual testing, where simulations are used rather than real world tests. As the number of potential evaluations are uncountable, scenarios are used as a guiding structure. Such a scenario defines the relevant static and dynamic parts of the vehicle's environment for example using the 6-Layer Model developed in the PEGASUS project [23]. Further, by focusing on dangerous scenarios a driving pilot can be tested more carefully in critical situations without risking property or lives [8,13].

However, creating tailored edge-cases for testing can be time consuming. In the last years, Large Language Models (LLMs) are used for several text-generation tasks. Used in vibe-coding, documentation and summarizing the abilities are used more and more in everyday tasks to save time and generate loads of text in a very short amount of time. By learning stochastic connections in language these models can generate many types of textual output. Using this ability to write test-scenarios could potentially save time and automatically create systematic test-cases tailored to the needs of the developers and safety engineers.

Given the enormous amount of code available on the internet (for example github.com), this data can be used to train LLMs and to create benchmarks to evaluate these models. Current benchmarks mostly evaluate the models through syntactic correctness and functional accuracy. Though there are still challenges in evaluating the models ability to generate efficient, readable and maintainable code [16,17,31].

Even in subjective and creative tasks like story-telling or legal writing the use of LLMs gains popularity. But given its subjective and inherent complexity, its validation do need expert supervision. Compared to code generation evaluating expert models on different domains need interdisciplinary experts in machine learning and for example legal expertise to ensure correctness [18,21].

Similar to generating working code in a programming language, a scenario can be generated in a given specification-language, like OpenDRIVE [4] by the Association for Standardisation of Automation and Measuring Systems (ASAM). Checking for the correct syntax can be easily achieved through the XML-specification, but checking for their semantic correctness is not done yet. To evaluate the content of the output the problem occurs that experts do need to check the correctness and plausibility, like in legal writing. Especially for never learned prompts, it is known that LLMs generalize poorly and can hallucinate or have difficulties with the semantic content and logical reasoning [15,28,31]. Therefore, it is important, that the output of LLMs is evaluated, such that, in our application, can be used to generate test scenarios.

One part of the semantic correctness is *normative correctness*, i.e., correctly adhere to a given set of norms, rules, or regulation. OpenDRIVE files need to follow a given set of rules defined by the standard. For example Schwab et al. [24] checked for gaps between the lanes at road crossings after recognising them in parametric road space models. The connection of geometric shapes (e.g. the roads), but also others geometric, topological or semantic rules are defined by OpenDRIVE rules. Until recently they were not checked at all. To guarantee the normative quality ASAM published the QC-Framework to check OpenDRIVE and OpenSCENARIO `XML`-data. `QC-OpenDRIVE` [6] uses this framework and implements some rules to ensure their correctness.

Generated OpenDRIVE-Scenarios by LLMs can be evaluated with the quality-checker. Key contributions of this paper are:

1. Integrating LLMs into a feedback-loop by evaluating generated outputs using `QC-OpenDRIVE` (Sect. 3),
2. Implementation of the rule E.5.9.1 (road.geometry.contact_point) for QC-Open-DRIVE (Sect. 4),
3. Evaluating correctness and content of generated outputs (Sect. 5).

2 Related Work

Evaluating code via benchmarks, like OpenAI's HumanEval [10] or Google Research's Mostly Basic Python Problems (MBPP) [7], is common practice. Some LLM-agents are also capable of using compilation errors of generated code to refine their output (e.g. GitHub Copilot agent mode [20]). But more subjective tasks, like evaluating the semantic correctness of different tasks, require experts to evaluate the model outputs [16–18,21].

The generation of concrete scenarios using LLMs is subject to prior works [9,22,32]. But none of these methods use a unified output syntax (e.g. OpenSCENARIO) to generate the data. This renders a standardized evaluation for LLM-generated scenarios much more complex or even impossible. Xiao et al. [29] describe scenarios in multiple logical steps, which could be used to evaluate the output by the logical definitions.

ASAM [3] develops standards for the development of autonomous driving systems. Members of the association are international car manufacturer, suppliers and research institutions. Use-cases like the development, testing and evaluation of driving systems are standardized by ASAM e.V. Therefor different data-formats, -models, protocols and interfaces are defined, establishing an easier interchange of data and tools [3,4].

ASAM OpenDRIVE [4] defines syntax and rules to describe road networks using Extended Markup Language (XML). OpenDRIVE is mainly concerned with describing the geometry of roads, lanes and objects like lane markings or signals. Such definitions can be based on real road-data or be synthetically generated.

Eisemann und Maucher [12,13] generate OpenDRIVE-Maps from LiDAR pointclouds. Segments of these points are combined and translated to OpenDRIVE. Becker et al. [8] define a logical description of a road-map, which is translated to OpenDRIVE. Both approaches do not use natural language processing.

Regarding the validation of OpenDRIVE, Schwab et al. [24] implemented simple rules, which mainly check for gaps between lanes. This uncovered gaps at road crossings in their OpenDRIVE data.

3 Integration of LLMs with QC-OpenDRIVE

The OpenDRIVE file format is based on Extensible Markup Language (XML). ASAM provides an XML schema that defines the structure of a valid OpenDRIVE file. The main OpenDRIVE document provides in-depth explanations about every aspect of the file format. It also defines a set of rules, that an OpenDRIVE file has to conform to. These rules do not cover syntactical correctness but aim to result in logical correctness of the described road network. E.g. rule E.5.9.5 (road.geometry.paramPoly3.length_match) requires that the actual curve length [of a road], as determined by numerical integration over the parameter range, should match [the parameter] @length [4]. Recently, ASAM published quality checkers for different standards, including OpenDRIVE. This framework called `QC-OpenDRIVE` [6] checks the syntax using the OpenDRIVE XML schema as well as a subset of the rules defined in the OpenDRIVE standard.

First experiments indicate that LLMs like GPT-4 can generate an OpenDRIVE file from simple prompts, when using the ChatGPT interface. But even the generation of very simple road networks composed of one or two roads often results in incorrect output. These errors can mostly be attributed to a violation of the OpenDRIVE XML schema, but also violations against rules of the OpenDRIVE standard do occur. It follows, that `QC-OpenDRIVE` should be able to detect such mistakes of an LLM. In such an event, the error report from `QC-OpenDRIVE` should help to find the faulty part of the generated OpenDRIVE and correct the mistake.

The idea of this feedback-loop is visualized in Fig. 1. Starting from a prompt to an LLM in natural language, an initial OpenDRIVE map will be generated and checked via `QC-OpenDRIVE`. In case of a detected error, the error report is given to the LLM to correct the mistake. This procedure can iteratively be repeated until all violations to the XML schema and the implemented rule set of `QC-OpenDRIVE` are eliminated. If not all mistakes can be corrected, the loop might either be terminated after a fixed number of iterations or when it is detected, that the same errors appear repeatedly. Using this approach, OpenDRIVE files can be generated automatically from natural language, while being syntactically correct and adhering to the implemented rules.

3.1 OpenDRIVE Generation Tasks for the Models

To identify the current abilities of different technologies and architectures in generating OpenDRIVE files, we tested three distinct models. They are given the same prompts of three tasks to generate simple road networks using a simple

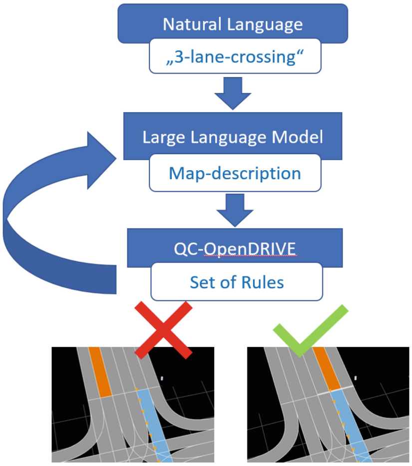


Fig. 1. Idea of a feedback-loop to correct invalid OpenDRIVE files generated by a LLM. The idea proposes to check its correctness by a given set of OpenDRIVE rules. With the error, the LLM hopefully can generate a correct output.

Chain-of-Thought (Cot) [27]. The validity of each output after one prompt is evaluated with the external tool QC-OpenDRIVE (see Fig. 1). The models are presented in Table 1 and represent state-of-the-art models leveraging Retrieval Augmented Generation (RAG) [19] or reasoning [11].

Table 1. List of used models for evaluation with their parameters and properties. Llama uses RAG, Qwen using reasoning and the mistral model uses none.

Model		Parameters	Reasoning [11]	RAG [19]
Mistral Large Instruct	[25]	123B	✗	✗
Meta Llama 3.1	[14]	70B	✗	✓
Qwen 3	[30]	235B	✓	✗

The chosen models are not fine-tuned to generate OpenDRIVE examples and are chosen to show the current abilities to correct its output with given feedback from QC-OpenDRIVE. As a baseline, the model Mistral Large [25] is chosen and used without any additional feature like reasoning or RAG. Meta's Llama 3.1 [14] features less parameters, but is using RAG to access the OpenDRIVE 1.7.0 documentation [2] as a PDF-file. Lastly, the most powerful tested model Qwen 3 [30] is trained to reason before giving the answer to the prompts. This model is added, as it may have been trained on some rules and might use the reasoning to validate the steps to make less errors. Each model was integrated into the feedback-loop with `QC-OpenDRIVE` to augment the models with the intended error correction capabilities.

We designed a small benchmark of three tasks, each composed of multiple prompts (see Table 2). To assist the models CoT is used. Wei et al. [27] have shown, that giving multiple step-by-step prompts can assists the models ability to generate more accurate responses. The first prompt in each task tests the capabilities of the models to generate an OpenDRIVE file purely from natural language. The additional prompts per task are intended to test the editing capabilities based on a already generated OpenDRIVE and an editing instruction. An editing instruction is only performed, if the previous prompt did not result in an error. Each task is generated in a new chat, so that the previous task is not in the context window.

Table 2. Prompts used for the different tasks. The system-prompt is the same for every model and task. Each prompt has a specific identifier to see the task and prompt id. For example the first task contains the three different prompts P1.1, P1.2 and P1.3. They are used in the same chat and used iteratively.

Task	ID	Prompt
System-prompt	-	You are a helpful assistant to create OpenDRIVE xodr-files. Be sure to use the correct schema-format. The output has to strictly follow the defined xml-schema - be sure to include the correct required attributes! But leave unnecessary elements and attributes out. For example leave the predecessor out, when there is no predecessor road. Use this header for the version 1.7.0 of OpenDRIVE: ‘‘‘ <?xml version="1.0" standalone="yes"?> <OpenDRIVE> <header revMajor="1" revMinor="7" name="" version="1.00" date="Wed Aug 14 11:25:56 2024" north="0.0000000000000000e+00" south="0.0000000000000000e+00" east="0.0000000000000000e+00" west="0.0000000000000000e+00"> </header> ‘‘‘ Resume the instructed content with the provided header.
Task 1	P1.1	Create a straight 10m road with a 2.5m wide lane in each direction.
	P1.2	Change the road, so that is 50m long.
	P1.3	Add one lane.
Task 2	P2.1	Create two roads. Each should have one left lane and one right lane.
	P2.2	Rotate one road for 180 degrees.
	P2.3	Add a link to connect both roads.
Task 3	P3.1	Create two roads. Each should have two left lanes and one right lane.
	P3.2	Add a link to connect both roads.

The chosen tasks include simple instructions to generate one or two roads, to modify and connect them. We currently refrain from more complex instructions, since experiments with different models have shown their inabilities in creating maps with three or more roads. We also noticed, that plenty of errors originate from an incorrect header in the generated OpenDRIVE files, which keeps meta information like the version or the global position of the map. To eliminate this error and focus at the content of the tasks, the header is provided as part of the system prompt.

4 Checking Normative Semantics

The rules from the ASAM standard OpenDRIVE are implemented in github.com/asam-ev/qc-opendrive/tree/main/qc_opendrive/checks. Since only a subset of all rules defined in the standard are currently implemented in `QC-OpenDRIVE`, an allegedly valid OpenDRIVE file might violate an unimplemented rule from the standard. And without an error report, such an invisible error cannot be corrected in the feedback-loop between an LLM and `QC-OpenDRIVE`. In our experiments, we often encountered a violation of rule E.5.9.1 (`road.geometry.contact_point`) by the LLMs. Thus, we implemented this rule to extend the rule set of `QC-OpenDRIVE`.

In `QC-OpenDRIVE`, the logic of a rule is implemented in the method `check_rule` of a new checker module. By registering this checker module in the `main.py`, the implemented method is automatically called when running `QC-OpenDRIVE`. For testing the implementation of a rule itself, it should also be tested using pytest. Test-`xodr`-files are stored in tests/data/ and added in the intended check-file in tests/.

4.1 OpenDRIVE Terminology

Before discussing rule E5.9.1 and its implementation in more detail, we need a rough understanding of the affected OpenDRIVE structures.

The geometry of the course of a road is defined in a `<planView>`. The line it follows is called the road reference line (see blue arrows in Fgure 2), from which the lanes of the road extend to the sides. Two roads can be connected with the `<link>` element, where the connection is defined by either the predecessor, when the other road connects to the start-point, or the successor, when the road connects to the end-point of the road reference line. The attribute *contactPoint* states, which end of the road reference line of the other road is connected to either the predecessor or successor point [1,4].

4.2 Rule E5.9.1: `road.geometry.contact_point`

The rule E.5.9.1 (road.geometry.contact_point) is performing a geometry check on a road definition. It states:

`Rule 5.9.1` If two roads are connected without a junction, the road reference line of a new road shall always begin at the `<contactPoint>` element of its successor or predecessor road. The road reference lines may be directed in opposite directions.

Each road defines its predecessor- and successor-elements in the `<link>` element of each connecting road. The point of contact is obligatory and given by the parameter *contactPoint*. The road reference line of both roads need to geometrically touch the end or start of the other contact point. With this rule, it has to be defined in the element, using the contact point `"end"` or `"start"`. The correct usage is illustrated in Fig. 2.

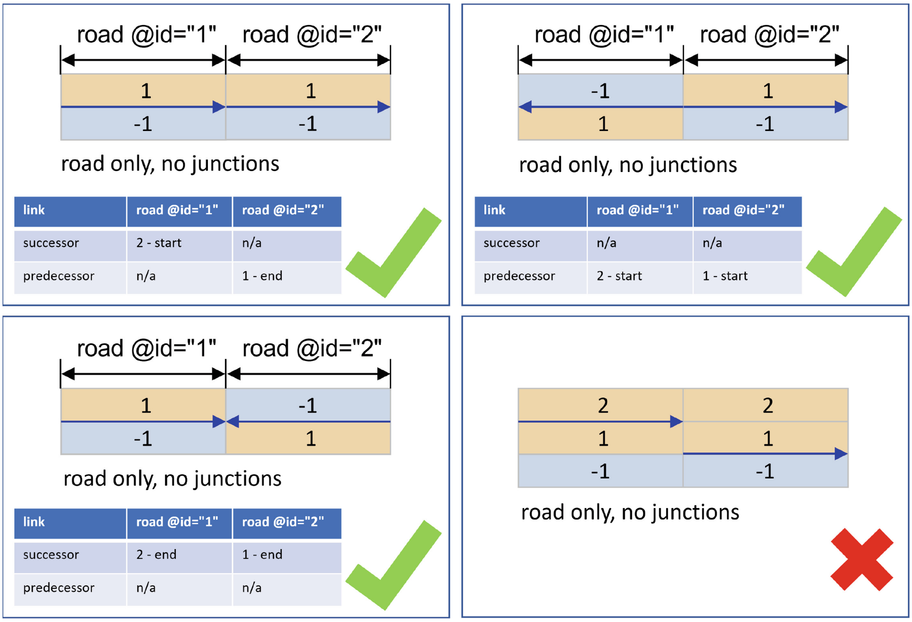


Fig. 2. Valid and invalid connections of contact-points from two roads. From OpenDRIVE [5].

In the upper left image the road with the ids “1” and “2” are connected. The successor link element of the road “1” is the road “2”, because the end of the road reference line, points at the start of the reference line of the road “2”. Because of this, the contact point of the successor of road “1” is the start of the road “2”. On the other side, the contact point of the predecessor from road “2” is the end of road “1”. When both road reference lines point to the opposite directions they are each other predecessors and have contact points at the start of each other. Lastly, when both road reference lines point to each other, they are successor elements with contact points at the end of each other. The bottom-right images illustrates the invalid case, when the contact points are not connected.

To check this rule, the geometric coordinates of the contact points and the start/end of the road reference lines have to be compared. To account for floating point errors, equality is assumed if the distance of coordinates is below a threshold value $\epsilon = 10^{-6}$. When the predecessor is checked, the start point of the current road reference line is used and when the successor is checked, it needs to be the end point. Depending on the parameter *contactPoint* of the `<link>` of the successor or predecessor, the contact point needs to equal the start or end point of the road reference line of the other road.

A pull-request with the implementation and tests is published for the official ASAM QC-OpenDRIVE github-repository. The implementation can be found at github.com/asam-ev/qc-open-drive/pull/126.

5 Evaluation

In the last section the models and prompts were presented. The details and tries are shown in Table 3. The table is ordered by the tasks. Mistrals model failed to generate the first prompt, so this model was not used for the further tasks.

Two of the three tested models, namely Qwen 3 and Llama 3.1, could use `QC-OpenDRIVE` as a feedback-loop. The mistral model could not fix its schema errors and did not return the corrected XML-output, while both other models were able to create correct OpenDRIVE files.

In the first task (P1.1, P1.2 and P1.3) the models were asked to create one road and modify it. Qwen 3 and Llama completed this task. Qwen 3 needed three attempts to fix the returned schema-errors. Llama 3.1 used RAG and finished the complete first task without errors. When investigating the output, it shows some minor differences to the intended task as it added more than one lane.

The second task (P2.1, P2.2 and P2.3) was to create two roads, rotate one and to connect both roads. Llama was not able to complete the task. In the first prompt it had one schema error, which could be fixed with the error message in the first try. After the second prompt it returned only the changes to the road and needed to be asked to return the full OpenDRIVE file. It could not fix the schema errors for the third prompt. Qwen 3 completed this task, while only having one error, missing the implemented rule E.5.9.1 (road.geometry.contact_point). Models might miss this rule and the resulted XML would be incorrect.

The last and third task was similar to the second one. Two roads have to be generated in one prompt. Both models needed three tries to generate a correct schema. As in the second task, Llama first printed only the changes, but after asking it to print the whole XML, the output was correct. Qwen 3 needed three tries for the second prompt. First the schema was invalid and in the second try the contact point of the link was invalid as in the second task. It could fix this error.

Using the feedback-loop both models could complete the tasks in a few minutes. Writing small OpenDRIVE files by hand would, depending on the knowledge, take similar or more time. But the strength of this models can be shown when different scenarios can be generated automatically. With the help of the feedback-loop, distinct scenarios can be described and then generated by the models.

Table 3. Validation results after each iteration (step) using different models for the generation of OpenDRIVE files in the predefined tasks. In each step a validation with `QC-OpenDRIVE` as well as a human inspection was performed and summarized here.

ID	Model	Step	`QC-OpenDRIVE` Validation	Human Inspection
P1.1	Mistral	1-6	Schema Error: invalid lane id	Alternates the same mistake for positive and negative lane-ids. Could not fix the error.
	Llama	1	No QC-OpenDRIVE Error.	Output as expected for the prompt.
	Qwen	1-2	Invalid schema	-
		3	No QC-OpenDRIVE Error.	Output as expected for the prompt.
P1.2	Llama	1	No QC-OpenDRIVE Error.	Two lanes with a width of 0cm on one side of the road reference line.
	Qwen	1	No QC-OpenDRIVE Error.	Output as expected for the prompt.
P1.3	Llama	1	No QC-OpenDRIVE Error.	Added three lanes. Two on the opposite driving direction.
	Qwen	1	No QC-OpenDRIVE Error.	Output as expected for the prompt.
P2.1	Llama	1	Schema Error.	-
		1	No QC-OpenDRIVE Error.	Output as expected for the prompt.
	Qwen	1	No QC-OpenDRIVE Error.	Output as expected for the prompt.
P2.2	Llama	1	-	Printed only the changed lines. Asked to print the whole XML again.
		2	No QC-OpenDRIVE Error.	Output as expected for the prompt.
	Qwen	1	No QC-OpenDRIVE Error.	Output as expected for the prompt.
P2.3	Llama	1-4	Schema Error	Could not fix the error.
	Qwen	1	Invalid rule road geometry contact point.	-
		2	No QC-OpenDRIVE Error.	Output as expected.
P3.1	Llama	1-2	Schema Error	-
		3	No QC-OpenDRIVE Error.	Output as expected for the prompt.
	Qwen	1-2	Schema Error	-
		3	No QC-OpenDRIVE Error.	Output as expected for the prompt.
P3.2	Llama	1	-	Printed only new roads. Manually asked again to print whole XML.
		2	No QC-OpenDRIVE Error.	Output as expected by the prompt.
	Qwen	1	Invalid schema.	-
		2	Invalid rule road geometry contact point.	-
		3	No QC-OpenDRIVE Error.	Output as expected for the prompt.

6 Conclusion

Natural language processing with LLMs might dramatically change many tasks typically done by humans. This includes describing legal writing, text summary generation, or even source code generation. Depending on the task and data availability, specialized experts need to evaluate the model, especially when data is sparse or the topic is of subjective nature. In automated driving, scenarios are a common tool for describing allowed and forbidden interaction of the vehicle and its environment (including vulnerable road users). Obtaining relevant scenarios covering the operational design domain is a huge challenge especially if they need to be handcrafted. In this paper, we investigate the automatic generation of road networks for such driving scenarios analogously to the generation of source code. We generate road networks as OpenDRIVE with different LLMs. By using OpenDRIVE, which is a standardized language, models can be compared and automatically checked for issues since a set of rules exists. We propose to integrate the LLM-based road network generation with automatic rule-checking in a feedback-loop in order to fix errors introduced by the LLM.

To demonstrate the feasibility of our feedback-loop, three LLMs are integrated with the open-source ASAM Quality Checker `QC-OpenDRIVE` for OpenDRIVE files, to iterative validate and correct the OpenDRIVE files generated by the LLMs. While most models have difficulties generating valid OpenDRIVE, by leveraging reasoning and RAG the tested models could generate syntactically valid road networks, especially when combined with `QC-OpenDRIVE` feedback. We show that syntactic schema errors as well as normative errors could be fixed using the feedback-loop. Unfortunately, not all rules of OpenDRIVE are yet implemented in `QC-OpenDRIVE`. Therefore, we extended `QC-OpenDRIVE` by implementing the rule E.5.9.1 `road.geometry.contact_point` which checks for road continuity at their contact points with other roads. This rule solved a major issue, we identified during our experiments. While Llama 3.1 failed to produce correct contact point in the second task and Qwen 3 in the third, they were able correctly connect roads given the feedback. Because LLMs have problems with implicit domain knowledge (e.g. the contact point), this does not only show practical utility in identifying invalid road connections as a critical requirement and the necessity of having validation tools but also shows the automatic feedback and correction is possible, leaving the creative part to the LLM and the corrective part to a rule checker.

The results show that human inspection is still needed, as there may be differences between the intended results and the generated output. In the future, the set of OpenDRIVE rules should be extended even further, as this will allow to correctly generate more complex OpenDRIVE files and allow automated tests with no human inspection. Moreover, this enables the training of specialized models and to ultimately create scenarios for testing autonomous driving systems in edge-cases. In the field of test-case design, such a automated LLM-based pipeline with a feedback-loop could enable hybrid approaches like using simulation and reinforcement learning to test edge-cases with minimal manual effort in validating the correctness and to meet industry standards.

By integrating LLMs with validation frameworks like `QC-OpenDRIVE`, this study paves the way for scalable, standardized, and semantically sound scenario generation, essential for the safe deployment of autonomous driving systems.

Acknowledgements. The research leading to these results is funded by the German Federal Ministry for Economic Affairs and Energy within the project "NXT GEN AI METHODS âĂŞ Generative Methoden für Perzeption, Prädiktion und Planung". The authors would like to thank the consortium for the successful cooperation.

References

1. ASAM: OpenDRIVE V1.8.1 (2024). https://www.asam.net/standards/detail/opendrive, https://publications.pages.asam.net/standards/ASAM_OpenDRIVE/ASAM_OpenDRIVE_Specification/latest/specification/03_terms_and_definitions/03_terms_and_definitions.html. Accessed 28 Apr 2025
2. ASAM e.V.: ASAM OpenDRIVE Version 1.7.0. https://www.asam.net/index.php?eID=dumpFile&t=f&f=4422&token=e590561f3c39aa2260e5442e29e93f6693d1cccd (2021). https://www.asam.net/index.php?eID=dumpFile&t=f&f=4422&token=e590561f3c39aa2260e5442e29e93f6693d1cccd. Accessed 27 May 2025
3. ASAM e.V.: ASAM Standards (2024). https://www.asam.net/. Accessed 28 Apr 2025
4. ASAM e.V.: OpenDRIVE V1.8.1 (2024). https://www.asam.net/standards/detail/opendrive. Accessed 28 Apr 2025
5. ASAM e.V.: OpenDRIVE V1.8.1/10 Roads/10.3 Road linkage/Figure 39 (2024). https://publications.pages.asam.net/standards/ASAM_OpenDRIVE/ASAM_OpenDRIVE_Specification/latest/specification/10_roads/10_03_road_linkage.html. Accessed 15 May 2025
6. ASAM e.V.: QC OpenDRIVE. https://github.com/asam-ev/qc-opendrive (2024), gitHub repository
7. Austin, J., et al.: Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021)
8. Becker, D., Ru , F., Geller, C., Eckstein, L.: Generation of complex road networks using a simplified logical description for the validation of automated vehicles. In: 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC), pp. 1–7 (2020). https://doi.org/10.1109/ITSC45102.2020.9294664
9. Chang, C., Wang, S., Zhang, J., Ge, J., Li, L.: LLMScenario: large language model driven scenario generation. IEEE Trans. Syst. Man Cybern. Syst. (2024). https://doi.org/10.1109/TSMC.2024.3392930
10. Chen, M., et al.: Evaluating large language models trained on code. CoRR abs/2107.03374 (2021)
11. Guo, D., et al.: DeepSeek-R1: incentivizing reasoning capability in LLMs via reinforcement learning (2025). DeepSeek-AI. https://arxiv.org/abs/2501.12948
12. Eisemann, L., Maucher, J.: Automatic odometry-less opendrive generation from sparse point clouds. In: 2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC), pp. 681–688. IEEE (2023). https://doi.org/10.1109/itsc57777.2023.10421842

13. Eisemann, L., Maucher, J.: Divide and Conquer: a systematic approach for industrial scale high-definition opendrive generation from sparse point clouds. In: 2024 IEEE Intelligent Vehicles Symposium (IV), pp. 2443–2450. IEEE (2024). https://doi.org/10.1109/IV55156.2024.10588602
14. Grattafiori, A., et al.: The llama 3 herd of models (2024). https://arxiv.org/abs/2407.21783
15. Huang, L., et al.: A survey on hallucination in large language models: principles, taxonomy, challenges, and open questions. ACM Trans. Inf. Syst. **43**(2), 1–55 (2025). https://doi.org/10.1145/3703155
16. Huynh, N., Lin, B.: Large language models for code generation: a comprehensive survey of challenges, techniques, evaluation, and applications (2025). https://arxiv.org/abs/2503.01245
17. Jiang, J., Wang, F., Shen, J., Kim, S., Kim, S.: A survey on large language models for code generation (2024). https://arxiv.org/abs/2406.00515
18. Lai, J., Gan, W., Wu, J., Qi, Z., Yu, P.S.: Large language models in law: a survey (2023). https://arxiv.org/abs/2312.03718
19. Lewis, P., et al.: Retrieval-augmented generation for knowledge-intensive NLP tasks (2021). https://arxiv.org/abs/2005.11401
20. Microsoft Corporation: Introducing Copilot Agent Mode (2025). https://code.visualstudio.com/blogs/2025/02/24/introducing-copilot-agent-mode
21. Pires, R., Junior, R.M., Nogueira, R.: Automatic legal writing evaluation of LLMs (2025). https://arxiv.org/abs/2504.21202
22. Ruan, B.K., Tsui, H.T., Li, Y.H., Shuai, H.H.: Traffic scene generation from natural language description for autonomous vehicles with large language model (2025). https://arxiv.org/abs/2409.09575
23. Scholtes, M., et al.: 6-layer model for a structured description and categorization of urban traffic and environment (2021). https://arxiv.org/abs/2012.06319
24. Schwab, B., Kolbe, T.: Validation of parametric opendrive road space models. Ann. Photogrammetry, Remote Sens. Spatial Inf. Sci. X-4-W2-2022 (2022). https://doi.org/10.5194/isprs-annals-X-4-W2-2022-257-2022
25. Team, M.A.: Mistral large. https://mistral.ai/news/mistral-large-2407 (2024). https://mistral.ai/news/mistral-large-2407. Accessed 26 May 2025
26. Wachenfeld, W., Winner, H.: The Release of Autonomous Vehicles, pp. 425–449. Springer, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-48847-8_21
27. Wei, J., et al.: Chain-of-thought prompting elicits reasoning in large language models (2023). https://arxiv.org/abs/2201.11903
28. Williams, S., Huckle, J.: Easy problems that LLMs get wrong (2024). https://arxiv.org/abs/2405.19616
29. Xiao, Y., Sun, Y., Lin, Y.: ML-SceGen: a multi-level scenario generation framework (2025). https://arxiv.org/abs/2501.10782
30. Yang, A., et al.: Qwen3 technical report (2025). https://arxiv.org/abs/2505.09388
31. Yang, J., et al.: Harnessing the power of LLMs in practice: a survey on chatgpt and beyond (2023). https://arxiv.org/abs/2304.13712
32. Zhang, J., Xu, C., Li, B.: ChatScene: knowledge-enabled safety-critical scenario generation for autonomous vehicles (2024). https://arxiv.org/abs/2405.14062

AGREE-Dog Copilot: A Neuro-Symbolic Approach to Enhanced Model-Based Systems Engineering

Amer Tahat(✉), Isaac Amundson, David Hardin, and Darren Cofer

Collins Aerospace, Cedar Rapids, IA 52498, USA
{amer.tahat,isaac.amundson,david.hardin,darren.cofer}@collins.com

Abstract. Formal verification tools like model checkers have long demonstrated their capability to ensure mission-critical properties are satisfied, yet their adoption in the aerospace and defense industries remains limited. Among the various reasons for slow uptake, difficulty in understanding analysis results (i.e., counterexamples) tops the list of multiple surveys. In previous work, our team developed AGREE, an assume-guarantee compositional reasoning tool for architecture models. Like many other model checkers, AGREE generates potentially large counterexamples in a tabular format containing variable values at each time step of program execution up to the property violation, which can be difficult to interpret, especially for novice formal methods users. In this paper, we present our approach for achieving explainable compositional reasoning using AGREE in combination with generative AI and we introduce AGREE-Dog, an open-source generative AI copilot integrated into the OSATE IDE. AGREE-Dog automates 16 DevOps and ProofOps steps, utilizing a novel context-selection and memory management system to efficiently manage evolving artifacts and historical interactions. We introduce structural and temporal metrics to evaluate the typically overlooked human contributions in generative AI-supported workflows. Evaluations using 13 UV fault-injection scenarios demonstrate a significant reduction in manual effort (less than 0.1 % of tokens authored by users), rapid convergence of counterexample repairs (84.6 % resolved in a single iteration, accuracy increasing to about 92 % after two iterations, and reaching 100 % within three iterations), and low LLM latency (average LLM response under 22 s, with negligible AGREE-Dog computational overhead). We also discuss limitations and future work. These promising results motivate further exploration into explainable model-based systems engineering (MBSE).

Keywords: LLM · Formal verification · MBSE · AGREE · AADL · Compositional reasoning

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 117–137, 2026.
https://doi.org/10.1007/978-3-032-07132-3_8

1 Introduction

Formal methods provide a mathematically rigorous means of verifying correctness in high-assurance systems, such as those used in the aerospace and defense industries. Certification guidance such as DO-333 [13] explicitly outlines how formal methods can meet airworthiness objectives for commercial aircraft software. Despite their proven effectiveness, adoption within traditional development workflows remains limited, hampered by scalability challenges, poorly designed tooling, and significant barriers to entry due to specialized training requirements [3].

The DARPA Pipelined Reasoning of Verifiers Enabling Robust Systems (PRO-VERS) program was launched to address these adoption barriers by developing scalable, human-centered formal verification workflows that seamlessly integrate into existing aerospace and defense engineering practices. Central to PROVERS' objectives is enabling usability even among engineers who lack extensive formal methods expertise, thereby fostering broader adoption and enhancing system dependability.

In response, our team has developed the Industrial-Scale Proof Engineering for Critical Trustworthy Applications (INSPECTA) framework [6]. INSPECTA comprises two integrated layers—*ProofOps* and *DevOps*—that embed formal verification directly into modern DevOps pipelines. The framework emphasizes scalability and explainability as primary design objectives, aligning closely with the PROVERS program's goals.

Within INSPECTA's *ProofOps* workflow, we employ the Assume-Guarantee Reasoning Environment (AGREE) [2], a compositional verification tool designed specifically for the Architecture Analysis and Design Language (AADL) [4]. Although AGREE avoids many of the scalability pitfalls found in monolithic verification tools, its counterexample outputs remain difficult to interpret. Like many model checkers, AGREE produces tabular counterexamples that trace the state of variables across multiple time steps. These can involve intricate temporal logic, nested states, and violations spanning architectural layers, posing challenges even for experienced engineers [7]. The diagnostic and repair process may span, hours, days, or weeks for large, evolving models based on user experties.

Recently, generative AI, and particularly large language models (LLMs), have shown promising potential to improve explainability and guide automated formal verification and counterexample repair. Early efforts include OpenAI's GPT-f, which achieved notable success in Metamath theorem proving [8,12]. Other initiatives have applied LLMs successfully to proof repair in Isabelle/HOL [5], theorem diagnosis in Coq [18], and discovering program invariants [11,17]. Stanford and VMware's Clover project represents another significant step forward, focusing on verifiable code generation with generative assistance [14]. Tahat et al. demonstrated high success rates using multi-turn conversational LLMs for proof repair in Coq, underscoring conversational learning's value in formal reasoning domains [15,16]. Apple's GSM-Symbolic [9] highlighted fundamental limitations of LLMs in symbolic reasoning tasks. Similarly, Amazon's recent SMT-backed hallucination prevention framework [1], while innovative, remains closed-source, available exclusively as a web service, and has yet to integrate within aerospace-specific MBSE pipelines such as those based on AADL.

We summarize the contributions of this paper as follows:

– We introduce AGREE-Dog, an open-source generative AI copilot integrated into the OSATE IDE[1], designed to automate and simplify the interpretation of counterexamples within aerospace and defense MBSE workflows using AADL and AGREE.
– An intuitive user interface coupled with detailed logging and traceability features, simplifying the typically challenging analysis of internal copilot interactions in generative AI-assisted systems.
– A context-selection and memory management algorithms that enhance prompt construction by triming irrelevant content, thereby reducing token usage, latency, cost, and enhancing recommendation accuracy.
– A novel set of structural and temporal evaluation metrics explicitly designed to quantify user effort, copilot automation, and interaction latency, capturing aspects often overlooked in evaluations of generative AI-assisted verification workflows.
– Experimental evaluations demonstrating AGREE-Dog's practicality, robustness, and effectiveness, validated through 13 diverse fault-injection test scenarios, highlighting rapid convergence of repairs and significant reduction of manual effort.

Throughout this paper, we provide simplified examples drawn from our test suite, illustrated clearly in the figures. We primarily focus on a simplified UV system described by the Car AADL package[2]. This open-source package, developed using AADL, contains several key subsystems—including top-level control, steering, and transmission—each specifying formal contracts verified using AGREE. The provided examples highlight AGREE-Dog's primary features, workflow, and practical advantages, demonstrating how the copilot supports users in interpreting counterexamples, identifying contract violations, and automating model repairs. Due to space limitations, a more detailed, end-to-end illustrative fault injection and repair scenario for the `Steering` subsystem is provided in Appendix A. Also, more comprehensive copilot interactions, detailed conversation samples, and log files are available in our GitHub repository[3].[4].

The paper is organized into seven key sections. Section 1 provides an introduction to formal verification and generative AI's role in enhancing model-based systems engineering. Section 2 introduces the AGREE tool, emphasizing explainability in compositional reasoning. Section 3 identifies core challenges motivat-

[1] https://github.com/loonwerks/AgreeDog.

[2] https://github.com/loonwerks/AgreeDog/tree/main/uploaded_dir/car/packages.

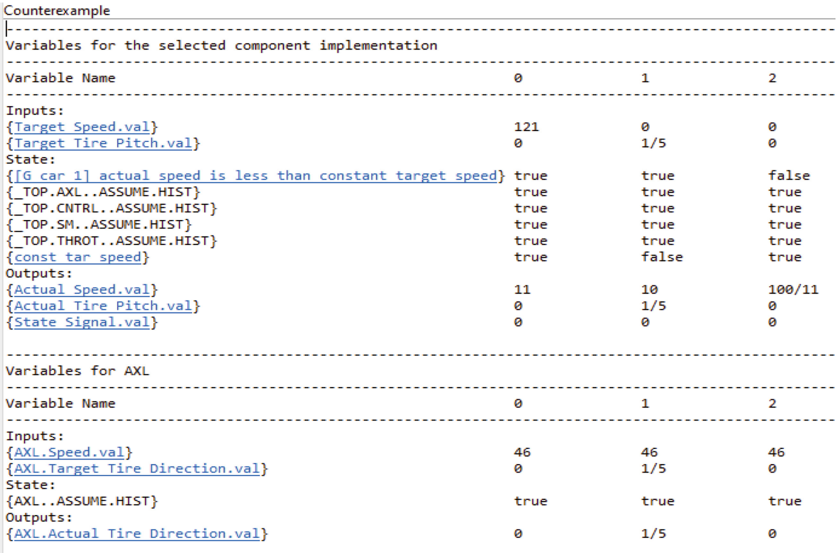
[3] https://github.com/loonwerks/AgreeDog/tree/main/logfiles-human-readble-conversations.

[4] https://github.com/loonwerks/AgreeDog/blob/main/shared_history/, such as, `conversation_history_20250427_215828.json`.

ing the development of AGREE-Dog. Section 4 details AGREE-Dog's architecture, including its user interface, backend workflow automation, and context-management algorithms. Section 5 describes novel structural and temporal metrics for evaluating human-in-the-loop interactions. Section 6 presents comprehensive experimental results demonstrating AGREE-Dog's efficacy using fault-injected scenarios. Finally, Sect. 7 concludes the paper, discussing limitations and avenues for future work.

2 Explainable AGREE

2.1 Overview



```
Counterexample
---------------------------------------------------------------------------------------
Variables for the selected component implementation
---------------------------------------------------------------------------------------
Variable Name                                                 0            1            2
---------------------------------------------------------------------------------------
Inputs:
{Target Speed.val}                                            121          0            0
{Target Tire Pitch.val}                                       0            1/5          0
State:
{[G car 1] actual speed is less than constant target speed}  true         true         false
{_TOP.AXL..ASSUME.HIST}                                       true         true         true
{_TOP.CNTRL..ASSUME.HIST}                                     true         true         true
{_TOP.SM..ASSUME.HIST}                                        true         true         true
{_TOP.THROT..ASSUME.HIST}                                     true         true         true
{const tar speed}                                             true         false        true
Outputs:
{Actual Speed.val}                                            11           10           100/11
{Actual Tire Pitch.val}                                       0            1/5          0
{State Signal.val}                                            0            0            0

---------------------------------------------------------------------------------------
Variables for AXL
---------------------------------------------------------------------------------------
Variable Name                                                 0            1            2
---------------------------------------------------------------------------------------
Inputs:
{AXL.Speed.val}                                               46           46           46
{AXL.Target Tire Direction.val}                               0            1/5          0
State:
{AXL..ASSUME.HIST}                                            true         true         true
Outputs:
{AXL.Actual Tire Direction.val}                               0            1/5          0
```

Fig. 1. This figure shows a code snippet of an AGREE-generated counterexample from the `Car` model. It illustrates a violation of the guarantee "G car_1: actual speed is less than constant target speed," which evaluates to false.

AGREE provides a formal contract language for specifying *assumptions* (i.e., expectations on a component's input and the environment) and *guarantees* (i.e., bounds on a component's behavior). Because AGREE is implemented as an AADL *annex* in the Open Source AADL Tool Environment (OSATE), the contracts are specified directly on components in the AADL model. AGREE then uses a k-induction model checker to prove properties about one layer of the architecture using properties allocated to subcomponents. The analysis proves correctness of (1) component interfaces, such that the output guarantees of each component must be strong enough to satisfy the input assumptions of downstream components, and (2) component implementations, such that the input assumptions of a system along with the output guarantees of its sub-components must be strong enough to satisfy its output guarantees.

When a contract violation is found (i.e., when an assumption is determined to be invalid or a guarantee is unsupported), AGREE produces a counterexample consisting of values for each system variable at each execution step. A sample counterexample is depicted in Fig. 1. Currently, OSATE includes the AADL Simulator tool that can accept an AGREE counterexample as input and walk through the trace in the graphical editor, but it is of limited help when it comes to identifying the root cause of the contract violation.

2.2 Making Counterexamples Actionable

We therefore desire AGREE counterexamples that are *actionable*; that is, an explanation of the violation in terms that will quickly lead to a passing analysis (e.g., by making changes to the formal contract or model). To achieve this, we implemented an interactive conversational copilot (AGREE-Dog) powered by GPT-4o and O3 multimodal generative AI models. It is specifically designed to assist AGREE users in identifying the root causes of counterexamples and applying targeted modifications during the model repair process, significantly reducing the turnaround time between verification attempts. The copilot is user-friendly and integrates seamlessly with the OSATE IDE (see Fig. 2) and (Fig. 3).

In the remainder of this paper, we explore the motivations that drove the development of AGREE-Dog, describe its key architectural features, and evaluate its effectiveness within representative modeling and verification workflows.

3 Motivations and Core Challenges

Drawing upon our practical experience integrating AGREE within MBSE workflows, in this section we highlight central challenges and key design principles that guided the development of our LLM-based solution for generating actionable counterexample explanations and facilitating automated model repairs.

3.1 Context-Aware Prompt Construction

AGREE-generated counterexamples typically involve numerous variables, intricate execution traces, and extensive AADL architectural data. Incorporating detailed LLM-generated code explanations and diagnostics exacerbates this challenge. Presenting these details directly to a generative AI model without careful management often result in excessive context size, increasing latency, hallucinations, costs, and potentially exceeding token limits. The key challenge is identifying and selecting only the most relevant context to include in prompts, ensuring accurate, concise explanations and actionable recommendations.

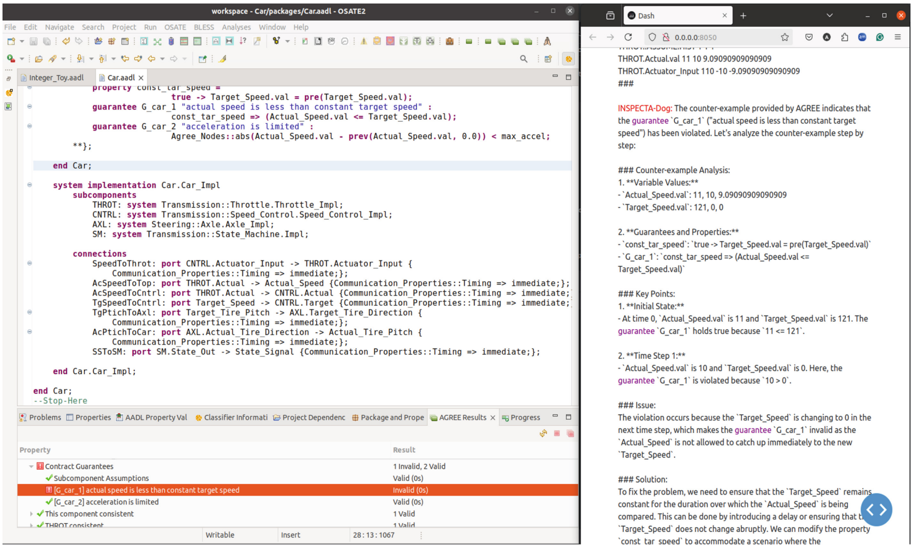


Fig. 2. AGREE-Dog copilot integrated within OSATE. This figure shows the user interface integration, with the OSATE IDE on the left and the AGREE-Dog copilot explanations intuitively displayed on the right. AGREE-Dog explains the root cause of the counterexample step-by-step in natural language, specifically, the violation occurs because Actual_Speed drops below Target_Speed, violating the guarantee G car_1.

3.2 Ensuring Validity of Automated Repairs

Generative models might propose repairs that, while plausible, could unintentionally violate established architectural interfaces or critical system properties. Maintaining consistency within compositional reasoning frameworks, such as AGREE, requires continuous validation. Thus, repairs must be tightly integrated with formal verification steps to ensure that each modification preserves overall system correctness.

3.3 Minimizing User Effort and Interaction Latency

Manually reviewing detailed logs and deeply nested temporal logic from counterexamples is both error-prone and time-consuming. An effective repair process must significantly reduce user overhead by automating log analysis, semantic comparisons between successive runs, and managing formal proof re-validation. Minimizing both system latency and human interaction time is essential to achieve an efficient, near-interactive model repair workflow.

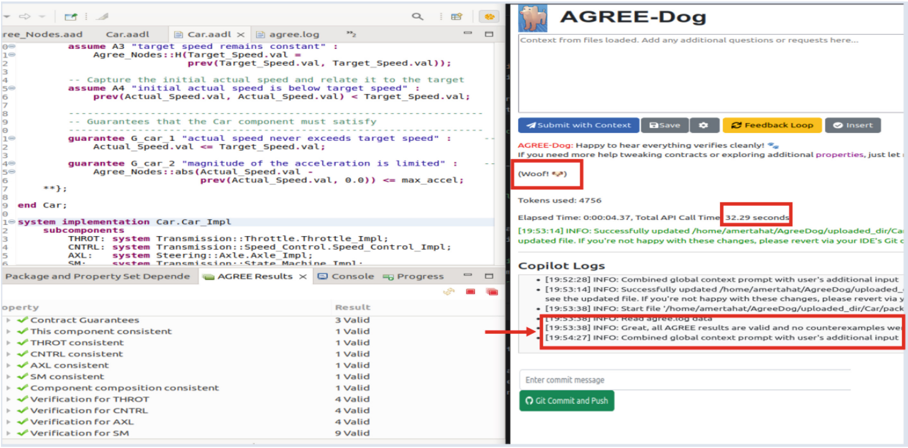


Fig. 3. AGREE-Dog UI interface showing integrated model diagnostics, user input, token count, response time, and push-button feedback loop. Each repair cycle is proof-aware and synchronized with AGREE log results.

4 AGREE-Dog Architecture and WorkFlow

This section details AGREE-Dog's main architectural components, and their roles in addressing usability and interpretability challenges identified previously (Sect. 3).

Specifically, we introduce AGREE-Dog's intuitive user interface, backend orchestration and workflow automation, optimized memory and context management algorithms, formal validation-driven feedback loops, and an internal logging subsystem supporting traceability and continuous refinement.

To illustrate how these components interact and integrate within a cohesive workflow, we refer the reader to Fig. 4. This figure highlights the interactions among the human user, OSATE IDE, AGREE formal verification tool, and AGREE-Dog's internal subsystems. Upon encountering a counterexample, the user interacts with the copilot through the provided interface, supplying hints and instructions. The copilot dynamically retrieves relevant context based on user input, IDE state, and formal tool feedback, employing an LLMAPI calls to produce intuitive explanations and suggest targeted model repairs. Internally, AGREE-Dog maintains detailed logs, recording key performance metrics. These metrics, introduced and defined explicitly in the following section, facilitate continuous improvement and future enhancements through fine-tuning based on high-quality interaction data.

4.1 User Interface

AGREE-Dog features an intuitive, streamlined user interface (UI), (Fig. 3), seamlessly integrated within the OSATE environment, designed specifically to

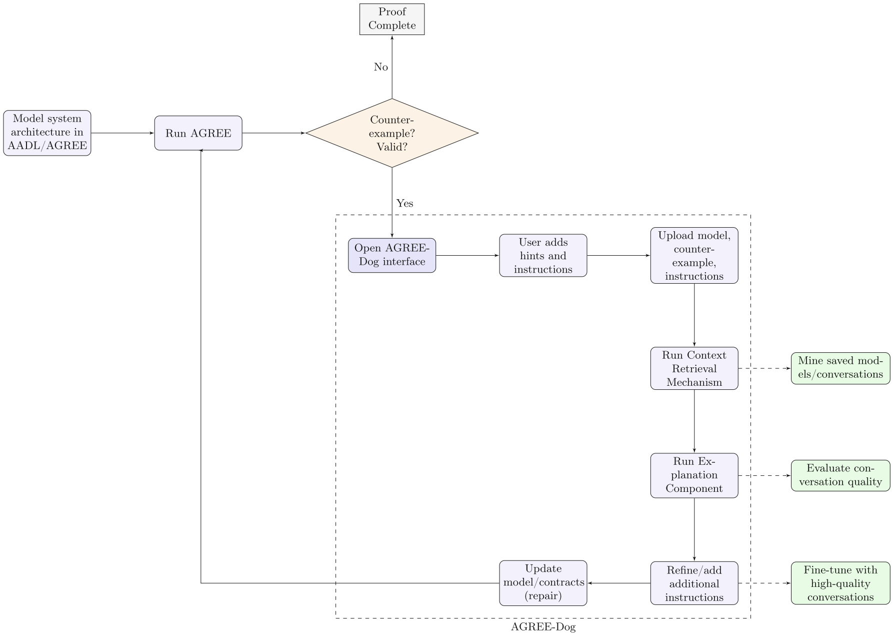


Fig. 4. AGREE-Dog workflow illustrating interactions among the user, IDE, formal tool, and copilot components, showing dynamic context retrieval, explanation generation, suggested repairs, and logging for continuous improvement.

minimize cognitive load and simplify complex verification tasks. Central to its usability are clearly labeled, push-button controls, enabling users to directly interact with counterexample explanations, formal validations, and system-level model repairs from a single coherent point of interaction.

A fundamental design principle of this UI is to balance transparency with abstraction—clearly presenting operational outcomes without burdening users with underlying complexities. This approach promotes efficiency, productivity, and verification effectiveness.

At the center of user interaction is the *Feedback loop* button, which synchronizes the internal state of OSATE with AGREE-Dog, with user inputs, updating its variables and internal data structures with most relevant context. This synchronization ensures coherence between AGREE-Dog's conversational state and the current OSATE project status, thus setting the stage for effective model analysis and refinement—detailed further in the next sections.

We complement this mechanism, with the *Insert* button which enables seamless integration of AGREE-Dog's suggested model repairs directly into OSATE, significantly streamlining what would otherwise be a tedious manual integration process. User-driven requests or specific instructions are submitted via the *Submit* button and can be further elaborated upon through an integrated conversational chat window. This conversational approach encourages precise, targeted refinements by enabling iterative and detailed guidance from the user.

Additional UI elements enhance interaction quality and knowledge retention. The *Save* button allows users to archive conversational histories for later review or further analysis and evaluations, as shown in Sect. 6, while the integrated *Git* control provides mechanisms for persistent storage, sharing of verification outcomes, and collaborative insight generation.

Moreover, advanced configurations are accessible via the dedicated *Settings* menu, allowing users to customize interaction workflows and select optimal LLM models tailored to specific tasks—such as generating explanations and repair suggestions (best supported by GPT-O3), or performing general-purpose, frequent tasks (ideally powered by GPT-4o)[5], as further detailed in Sect. 4.2.

4.2 Backend Function Call Graph and Workflow Automation

To support interactive workflows, AGREE-Dog automates 16 critical DevOps and ProofOps steps. The backend orchestration, summarized in Appendix A (Fig. 7), manages operations ranging from artifact selection and prompt construction to automated AGREE invocations. AGREE-Dog utilizes context and history-aware agents that dynamically select relevant artifacts, perform semantic diffs, and invoke proof engines. Each backend operation is highly optimized, incurring negligible runtime overhead (less than one second per operation), as demonstrated by the empirical results in Sect. 6.

4.3 Context Selection and Memory Management Optimization

Effective context selection and memory management are critical to AGREE-Dog's ability to provide precise explanations and actionable repairs involving complex AADL artifacts, execution traces, and user instructions. Addressing these challenges requires the sophisticated, carefully optimized mechanisms embedded within AGREE-Dog's core copilot algorithm.

Core Copilot Algorithm. Algorithm 1 embodies the central context management strategy of AGREE-Dog, as conceptually outlined in Fig. 4. This algorithm integrates intelligent conversational state tracking, dynamic artifact selection, and optimized memory management processes to efficiently support model verification and repair tasks.

Optimized Dynamic Context Retrieval and Updates. **Algorithm** 1 dynamically selects a minimal yet sufficient context—including relevant AADL source files, counterexamples, AGREE logs, and system requirements, and interactive user

[5] At the time of writing, OpenAI recommends GPT-O3 for reasoning-intensive tasks and explanations, offering better reasoning performance but slightly higher latency. In contrast, GPT-4o is optimized for general-purpose tasks, providing lower latency.

instructions—for accurate verification and effective repair interactions. Leveraging its integrated dynamic Context Retrieval component, the algorithm selectively imports only the most recently updated model artifacts, identified through AGREE-log updates received from OSATE, by traversing dependency chains and referencing stored conversational data.

By default, the context retrieval strategy excludes standard training data such as core libraries typically present in LLM training sets, thus optimizing token usage. However, users retain flexibility to explicitly include or exclude any files from the complete import chain during initialization, incorporating selected context elements into the initial prompt. Once included, these explicitly imported files remain static in memory unless updated explicitly by the user or signaled via AGREE logs. Additionally, natural-language requirement files (e.g., CSV-based inputs), not tracked by AGREE logs, are monitored independently with automatic checks performed every two seconds to detect changes.

This nuero-sympolic (intersymbolic) and user-customizable selection process significantly reduces redundancy, enhances convergence speed toward correct model solutions, minimizes generative model latency, and mitigates hallucinations caused by irrelevant context.

Memory Management Optimization Mechanism. A critical component of Algorithm 1 is its internal conversational memory management subsystem, detailed fully in Appendix A. This subsystem employs a structured, list-based representation to balance immediate responsiveness with longer-term conversational persistence. Short-term interactions are retained in readily accessible memory for efficient prompt updates, while less immediate interactions can optionally be saved locally by the user or systematically migrated into persistent storage managed by integrated Git version control. This approach allows AGREE-Dog to effectively recall prior repair strategies and interaction histories, thus enhancing iterative repairs and significantly reducing the overhead associated with manual snapshots management.

Furthermore, AGREE-Dog's memory management strategy directly facilitates ongoing system refinement. Archived conversational histories and validated repairs can subsequently be leveraged to fine-tune the underlying generative models, enabling continual improvement in the quality of explanations and repair suggestions.

4.4 Verification-Aware Feedback Loop and Repair Validity

AGREE-Dog's neuro-symbolic reasoning, achieved by combining AGREE's formal verification with generative AI explanations, establishes a rigorous, verification-aware repair loop. Central to this process, AGREE-Dog invokes AGREE externally via API calls to ensure that all proposed repairs strictly adhere to system-wide consistency and soundness criteria.

This verification-integrated approach not only acts as a safeguard against unsound or logically inconsistent model modifications but also enhances the quality of data fed into the generative model. By proactively filtering invalid

Algorithm 1: AGREE-Dog Interactive Copilot Prompt Construction and Counterexample Handling

Input: AADL Model Files, Counterexample File (optional), System Requirements (optional)
Output: Prompt for GPT-based AGREE-Dog Copilot, Actionable Repair Suggestions
Initialization:
Load command-line arguments: working directory, start file, counterexample, requirements file;
Load OpenAI API key;
Initialize logging system;
Main Procedure:
if *requirement file provided* **then**
    Load and include requirements in prompt context;
else
    Set requirements context to *"No sys_requirement file provided"*;
Prompt Construction:
Read top-level AADL file from provided workspace;
Parse import chain and extract relevant AADL files (avoid standard libraries);
if *counterexample provided (CLI or file)* **then**
    Load counterexample into context;
else
    Search for recent counterexamples:
    – Check command-line provided counterexample path first.
    – If unavailable, parse `agree.log` for failing contracts.
    – Match failing contracts with available counterexample XML/text files.
    – Extract and format counterexample(s) for inclusion.

Construct comprehensive prompt with:

1. System Requirements (if available)
2. AADL Model Content
3. Counterexample(s) Explanation
4. Explicit instructions for GPT (repair suggestions within AADL syntax)

Interaction and Feedback Loop (via Dash UI):
while *copilot session active* **do**
    Receive additional user input (optional);
    Combine with the current prompt context (if any);
    Submit prompt to GPT-4o/GPT model via OpenAI API;
    Retrieve response:

    – Explain verification failures clearly
    – Suggest repairs in AADL syntax, respecting requirements

    Present GPT response to user;
    Log interaction and update metrics (latency, tokens used, etc.);
    if *user applies modifications* **then**
        Extract AADL repair suggestions from GPT response;
        Safely overwrite the original AADL model file;
        Notify user of successful update or handle exceptions;
Quality Assessment and Logging:
Automatically record metrics (timestamps, token use, latency);
Store interaction logs for future analysis and fine-tuning;
Shutdown Procedure:
On user request, terminate the copilot session gracefully;

suggestions, AGREE-Dog reduces the overall token volume required, thereby significantly improving LLM latency and maintaining model reliability and trustworthiness. Such integration distinctly differentiates AGREE-Dog from purely neural LLM approaches, which inherently lack logical soundness checks and may erroneously group logically distinct, yet superficially similar elements [9,15,16].

Additionally, the semantic diffing mechanism embedded in AGREE-Dog detects relevant model changes precisely across iterative repair cycles, facilitating faster convergence to formally valid solutions. This integrated neuro-symbolic loop thus effectively bridges generative AI capabilities with rigorous MBSE based formal verification.

4.5 Traceability, Logging, and Continuous Refinement

The extensive logging within AGREE-Dog serves dual purposes. First, it facilitates real-time diagnostics, enabling rapid identification of effective conversational interactions and successful repair strategies. As illustrated in the AGREE-Dog user interface (Fig. 3), key performance indicators—including AGREE validity status, token count, system and human return time, and LLM latency—are prominently displayed, providing users immediate feedback to gauge interaction effectiveness.

Second, the detailed logs support ongoing system refinement by highlighting conversational patterns consistently associated with high-quality, formally valid repairs. This capability directly informs the metrics employed for evaluating AGREE-Dog's performance, as further detailed in Sect. 6 and Sect. 5. By analyzing logged interaction timelines and human response metrics, AGREE-Dog identifies optimal repair strategies, promotes knowledge reuse, and reduces manual intervention, significantly enhancing both short-term repair efficiency and long-term knowledge retention.

5 Evaluation Metrics

This section introduces the core metrics used to evaluate AGREE-Dog's performance. We organize them into two complementary categories: structural (or spatial) metrics, which quantify the shape and volume of interaction, and temporal metrics, which capture responsiveness and turnaround time. Together, these metrics enable a holistic assessment of automation, effort, and cost.

5.1 Structural Metrics

Structural metrics quantify how the repair process unfolds—how many interactions occurred, how much human input was required, and how much computational effort was expended.

Total Token Count (TTC). This metric captures the total number of tokens exchanged during a repair conversation, including both human-authored tokens and those generated by AGREE-Dog—either by the LLM or by the system's prompt constructor:

$$\text{TTC} = \text{Human Tokens} + \text{AGREE-Dog System Tokens} \tag{1}$$

TTC serves as a proxy for computational and financial cost (e.g., token-based billing), independent of who authored the tokens. However, it does not by itself distinguish the extent of human involvement.

Human Input Ratio (HpR). This metric measures the proportion of human-authored tokens relative to the total token count:

$$\mathrm{HpR} = \frac{\text{Human-Authored Tokens}}{\text{Total Tokens in Conversation}} \tag{2}$$

A lower HpR suggests higher automation, with the system contributing more heavily to the conversation. When considered with TTC, this helps differentiate brief, efficient sessions from those with more human effort or verbosity.

Number of Repair Cycles (N_{RC}***)***. This metric counts the number of conversational cycles required to reach a valid system state:

$$N_{\mathrm{RC}} = \text{Number of Repair Cycles Until Validity} \tag{3}$$

Each cycle begins with a `start_file_read` message and ends with a `validity_status: valid` confirmation. Together, N_{RC}, HpR, and TTC form a triplet that reflects the intensity, automation level, and computational cost of the repair process.

Repair Success Rate (RSR). This metric measures how often AGREE-Dog succeeds in exactly N_{RC} cycles:

$$\mathrm{RSR}(N_{\mathrm{RC}}) = \frac{\text{Number of Tests Solved in } N_{\mathrm{RC}} \text{ Cycles}}{\text{Total Number of Tests}} \tag{4}$$

Cumulative Repair Success Rate ($\mathrm{RSR}_{\mathrm{acc}}$***).*** This cumulative variant captures the percentage of tests solved within a given number of cycles:

$$\mathrm{RSR}_{\mathrm{acc}}(N_{\mathrm{RC}}) = \frac{\text{Number of Tests Solved in } \leq N_{\mathrm{RC}} \text{ Cycles}}{\text{Total Number of Tests}} \tag{5}$$

These success rate metrics extend the basic structural measures to account for convergence and consistency. They are operationalized in Sect. 6, where we analyze repair outcomes and cycle distributions (see Fig. 6).

5.2 Temporal Metrics

While structural metrics describe what happened during the interaction, temporal metrics quantify how long it took—enabling assessments grounded in real-world engineering effort and user experience.

Wall-Clock Time (WCT). The total elapsed time from the first user input to final validation. WCT serves as a practical proxy for engineering effort and turnaround time. Shorter durations may reflect both efficient execution and the usefulness of AGREE-Dog's guidance.

WCT also conveys a notion of ***Repair Speed***—how many valid tasks are completed per unit of time. For example, in our evaluation (Sect. 6), the mean

WCT per valid cycle was 2 min and 9 s, with a median of 1 min and 39 s.

LLM Latency. The average LLM response time per repair cycle. Lower latency improves interactivity and helps maintain user focus, especially in iterative or multi-step sessions.

Next, we define a dependent metric based on the previous temporal measurements to estimate the human return time.

Human Return Time (HRT). This metric estimates the time required for a human to return to a task and make cognitively informed decisions necessary to reach validity during the interaction. It is calculated as the total wall-clock time minus the time AGREE-Dog spends in CPU execution and large language model (LLM) processing. Formally:

$$\text{HRT} = \text{Wall-Clock Time} - \text{CPU Time} - \text{LLM Response Time} \tag{6}$$

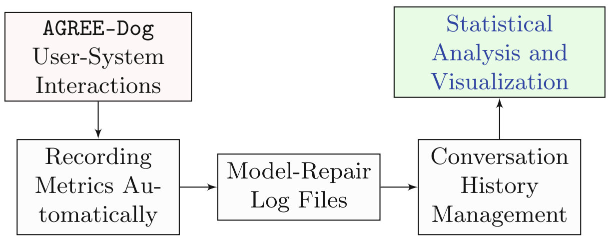


Fig. 5. AGREE-Dog Conversation Quality Assessment Workflow (CQAW). The workflow tracks structural metrics from conversation histories and temporal metrics from copilot logs, leveraging timestamps to measure user and LLM response latencies. Finally, metrics are analyzed and visualized using AGREE-Dog's statistical utility.

5.3 Composite Score: Structural and Temporal Dimensions

To facilitate comprehensive evaluation, we interpret AGREE-Dog's performance using a composite score that integrates both structural and temporal dimensions:

$$(N_{\text{RC}}, \text{HpR}, \text{TTC}, \text{Wall-Clock Time}, \text{LLM Response Latency}, \text{CPU Time}) \tag{7}$$

This composite vector captures not only the automation level and conciseness of each repair session but also temporal efficiency. For instance, sessions with identical token counts and automation levels might still differ significantly in usability due to variations in latency or total duration. Additionally, this formulation supports the calculation of derived metrics, such as Human Return Time (HRT) (Eq. 6) and Repair Success Rate (RSR).

By combining structural and temporal perspectives, the composite score provides nuanced insights into humanâĂŞsystem interaction dynamics, balancing token efficiency with practical engineering outcomes.

6 Experimental Evaluation

6.1 Evaluation Setup and Fault Injection Protocol

Using the Conversation Quality Assessment Workflow (CQAW, Fig. 5), we systematically tracked structural and temporal metrics to comprehensively evaluate AGREE-Dog. Our experiments involved thirteen fault-injected test scenarios based on an AADL-based Car model. Each scenario featured dynamically evolving artifacts—including AADL source files, natural-language requirements, counterexample traces, AGREE log files, and LLM-generated diagnostics—culminating in approximately 32,100 tokens across all scenarios. On average, scenarios began with around 400 lines of AADL and log content, fewer than 100 lines of counterexample traces, and less than 100 lines of natural-language inputs.

Table 1. Summary of Structural and Temporal Metrics for AGREE-Dog Evaluation

Metric	Result
Structural Metrics	
System Validity	100% achieved for all test scenarios
Repair Success Rate (RSR)	11/13 (84.6%) in 1 cycle; 1/13 in 2 cycles; 1/13 in 3 cycles
Human Input Ratio (HpR)	$< 0.1\%$ of total tokens
AGREE-Dog Generated Input	$> 99.9\%$ of total tokens
Token Use (per test suite)	4.8k, 5.5k, 22k tokens
Temporal Metrics	
Wall-Clock Time (WCT)	Mean: 2:09 min; Median: 1:39 min
LLM Latency (per cycle)	Mean: 22 s; Range: 4–33 s

Faults targeted three safety-critical subsystems (Top-Level Control, Steering, and Transmission), triggering 16 repair cycles. Injected faults covered typical behavioral and contract-level violations—ranging from incorrect assumptions, logic errors, and range violations to faulty assignments and temporal inconsistencies. Repairs were accepted only after passing AGREE's formal verification and manual user confirmation via AGREE-Dog's `insert` command, ensuring both correctness and soundness.

Evaluation Metrics. Table 1 summarizes AGREE-Dog's structural and temporal performance metrics (defined in Sect. 5). Figure 6 visualizes repair convergence across the scenarios.

Next, we summarize the key insights obtained from our evaluation, supported by quantitative data presented in Table 1 and visualized in Fig. 6.

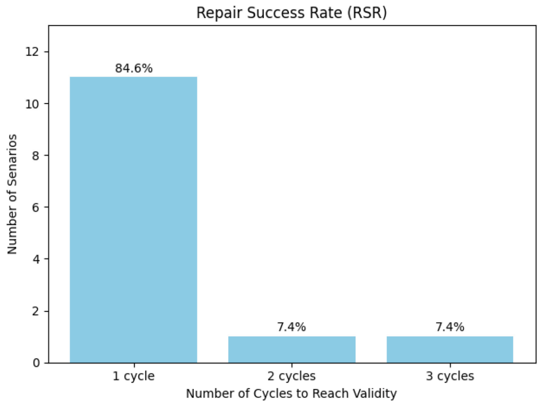


Fig. 6. Repair cycles required by AGREE-Dog to achieve system-wide validity.

6.2 Key Results

This evaluation demonstrates the feasibility of integrating generative AI (GenAI) with formal verification in Model-Based Systems Engineering (MBSE). By combining large language model reasoning with AGREE-based validation in OSATE, AGREE-Dog delivers verifiable repairs with minimal human effort.

1. **Rapid Convergence with Reduced Human Intervention Frequency:** AGREE-Dog resolved approximately 85% (11 out of 13) of the test cases within a single cycle, while the remaining cases required two or three cycles (approximately 7.5% each). This demonstrates swift convergence and significantly reduces the frequency of user interventions needed across diverse fault scenarios.
2. **High Automation with Minimal Human Effort:** Estiamted by (HpR) metric, Human-generated content constituted less than 0.1 % of the overall tokens, with AGREE-Dog autonomously generating more than 99.9 % via its integrated prompt construction mechanism and language model. Combined with the rapid convergence rate noted previously, this outcome highlights AGREE-Dog's capability to effectively automate model repairs, significantly reducing manual input relative to the extensive verification contexts encountered.
3. **Efficiency and Reduced Human Return Time (HRT):** AGREE-Dog demonstrated significant computational and cognitive efficiency throughout the evaluation. Internal computational overhead consistently remained below one second per operation, complementing an average LLM latency of approximately 22 s per cycle. While the median overall wall-clock time (WCT) was about 1 min and 39 s the average human response time (HRT) was approximately 1 min and 3 s. This average, however, was notably skewed by two outlier cases; in fact, 85 % of scenarios achieved total resolution (WCT) in under 45 s—including LLM latency—limiting human analysis and decision-making time to less than 23 s per scenario in 11 out of 13

cases. Compared to traditional manual verification approaches, which typically require hours or days, AGREE-Dog's structured guidance and intuitive natural-language explanations significantly reduced human cognitive effort estimated by (HRT) metric and the overall interaction duration (WCT).

7 Conclusions and Future Work

To enhance the explainability and usability of AGREE-generated counterexamples, we developed AGREE-Dog, the first open-source conversational copilot specifically integrating neuro-symbolic methods with AGREE's formal verification tools within the OSATE environment. AGREE-Dog produces intuitive, natural-language explanations for complex counterexamples, significantly reducing human effort and cognitive load required for formal model repairs. Our experimental evaluation demonstrates AGREE-Dog's feasibility and effectiveness at realistic MBSE scales—handling scenarios spanning tens of thousands of tokens without notable performance degradation. These initial results provide a promising evidence for the practical utility and scalability of neuro-symbolic methods, highlighting significant potential for broader educational and industrial adoption. AGREE-Dog is publicly accessible on GitHub.

Despite these encouraging outcomes, several avenues for future improvement and exploration remain. We intend to continue evaluating AGREE-Dog on increasingly sophisticated and complex system models and formal specifications.

Furthermore, ongoing developments in large-context language models (e.g., GPT-4.1's 1-million-token context window) offer substantial opportunities to explore more autonomous decision-making frameworks, including reinforcement learning-driven judge-router-worker agentic architectures. Such systems could dynamically and autonomously select optimal repair strategies, further reducing manual intervention. Additionally, extending AGREE-Dog's capabilities to emerging modeling standards, such as SysML v2 [10], represents a key future goal.

Lastly, the integration of our evaluation workflow into INSPECTA's DevOps Assurance Dashboard will facilitate continuous monitoring, displaying metrics such as model modifications, counterexample handling efficiency, and AGREE usage statistics. This integration aims to quantify the tangible benefits of more explainable counterexamples, driving targeted improvements in usability and overall user experience.

We look forward to exploring these directions in future work and reporting further advancements toward integrating neuro-symbolic verification approaches in MBSE.

Acknowledgments. This work was funded by DARPA contract FA8750-24-9-1000. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Algorithm 2: Memory Management and Prompt Optimization in AGREE-Dog

Input : User input, conversation state, AADL model repository, optional requirements file
Output: Optimized prompt, updated conversation history
Initialize Short-Term, Temporary, and Long-Term memories;
Identify and load recently updated files:
- Identify recently updated files in repository.
- Load **only** these updated files into Temporary memory.
- Cache filenames and timestamps.

Integrate system-level requirements (if provided);
Construct prompt from:

- Updated files from Temporary memory.
- User input and interaction history.
- System-level requirements.

Ensure prompt size within token limits (truncate oldest entries if necessary);
Generate response from AGREE-Dog model;
Update Short-Term memory with latest interaction;
if ***User selects*** *Save Conversation* **then**
    Save conversation to Long-Term memory;
if ***User selects*** *Commit to Git* **then**
    Stage conversation and updated files;
    Commit and push to remote repository;
return optimized prompt, updated conversation history;

A Appendix

- **Initial Axle specification:**[6]

```
guarantee G_axle_1 "roll limiter":
    \begin{verbatim}
guarantee G_axle_1 "roll limiter":
  if (Agree_Nodes::abs(Target_Tire_Direction.val) > 0.20
      and Speed.val > 45.0)
  then
    Actual_Tire_Direction.val = 0.20
  else
    Actual_Tire_Direction.val = Target_Tire_Direction.val;
```

- **Fault injection:** Introduced by changing the guarantee threshold from `0.20` to `0.10`:

[6] The full model is available at: https://github.com/loonwerks/AgreeDog/blob/main/uploaded_dir/car/packages/Steering.aadl.

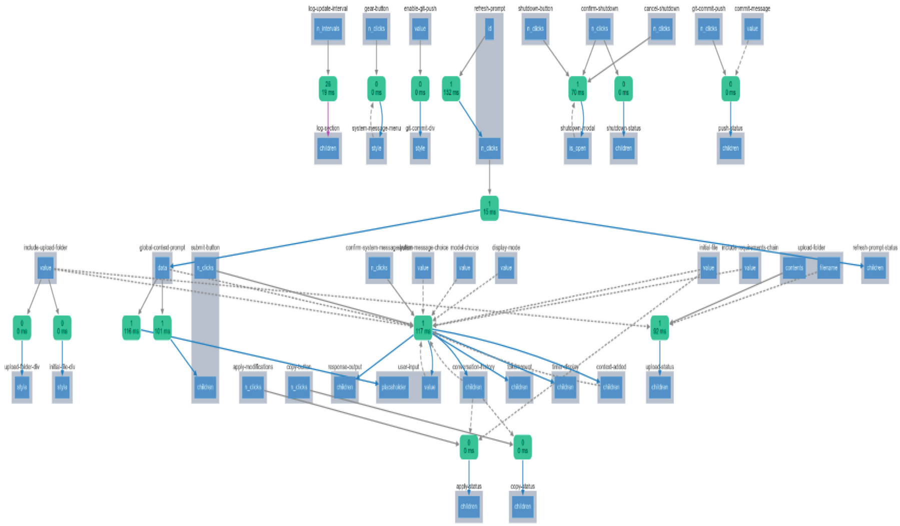


Fig. 7. AGREE-Dog backend function call graph illustrating automated DevOps/ProofOps orchestration. Nodes represent key operations, while edges indicate dependencies and data flows between components.

```
if (Agree_Nodes::abs(Target_Tire_Direction.val) > 0.10
    and Speed.val > 45.0)
```

- **Counterexample detection:** AGREE immediately detected a counterexample, indicating the specification and implementation mismatch due to the altered guarantee.
- **AGREE-Dog first iteration:** The copilot quickly located the faulty threshold and corrected it back to `0.20`, but inadvertently removed the `else` statement from the guarantee:

```
else
  -- Otherwise pass the request straight through
  Actual_Tire_Direction.val = Target_Tire_Direction.val;
```

- **Second iteration (Diff mechanism):** A subsequent AGREE run revealed another counterexample caused by the missing `else` statement. The user then guided AGREE-Dog to utilize the built-in diff mechanism, comparing the new and old files. After clearly identifying the missing `else` clause, the user instructed the copilot explicitly to reinsert it, resolving the issue completely.
- **Resolution and insights:** After the second iteration, the corrected model passed formal verification successfully. This example emphasizes the critical role of symbolic verification and human feedback in conjunction with neuro-symbolic copilots. It highlights the risks associated with relying solely on neural approaches without rigorous symbolic validation.

References

1. AWS News Blog: Prevent factual errors from llm hallucinations with mathematically sound automated reasoning checks (preview). AWS Blog Post(2024), Accessed 11 May 2025
2. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 126–140. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_13
3. Davis, J.A., et al.: Study on the barriers to the industrial adoption of formal methods. In: Pecheur, C., Dierkes, M. (eds.) Formal Methods for Industrial Critical Systems, pp. 63–77. Springer, Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-41010-9_5
4. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley Professional, 1st edn. (2012)
5. First, E., Rabe, M.N., Ringer, T., Brun, Y.: Baldur: Whole-proof generation and repair with large language models. arXiv preprint arXiv:2303.04910 (2023)
6. INSPECTA Project: Inspecta. https://loonwerks.com/projects/inspecta.html, Accessed: 19 May 2025
7. Kaleeswaran, A.P., Nordmann, A., Vogel, T., Grunske, L.: A systematic literature review on counterexample explanation. Inform. Softw. Technol. **145** (2022)
8. Megill, N., Wheeler, D.A.: Metamath: A computer language for mathematical proofs (2019)
9. Mirzadeh, S.I., Alizadeh, K., Shahrokhi, H., Tuzel, O., Bengio, S., Farajtabar, M.: GSM-symbolic: understanding the limitations of mathematical reasoning in large language models. In: The Thirteenth International Conference on Learning Representations (2025). https://openreview.net/forum?id=AjXkRZIvjB
10. Object Management Group (OMG): Systems modeling language (sysml) v2 specification. (2024), Accessed 19 May 2025
11. Pei, K., Bieber, D., Shi, K., et al.: Can large language models reason about program invariants? Proceedings of the 40th International Conference on Machine Learning (July 2023)
12. Polu, S., Sutskever, I.: Generative language modeling for automated theorem proving. arXiv preprint arXiv:2009.03393 (2020)
13. RTCA: DO-333: Formal Methods Supplement to DO-178C and DO-278A (December 2011)
14. Sun, C., Sheng, Y., Padon, O., Barrett, C.: Clover: Closed-loop verifiable code generation. arXiv preprint arXiv:2310.17807 (2024)
15. Tahat, A., Hardin, D., Petz, A., Alexander, P.: Metrics for large language model generated proofs in a high-assurance application domain. In: High Confidence Software and Systems Conference (HCSS 2024) (2024)
16. Tahat, A., Hardin, D., Petz, A., Alexander, P.: Proof repair utilizing large language models: A case study on the copland remote attestation proofbase. In: Proceedings of International Symposium On Leveraging Applications of Formal Methods Verification and Validation (AISolA) (2024)
17. Wu, H., Barrett, C., Narodytska, N.: Lemur: Integrating large language models in automated program verification. arXiv preprint arXiv:2310.04870 (2023)
18. Zhang, S., First, E., Ringer, T.: Getting more out of large language models for proofs. arXiv preprint arXiv:2305.04369 (2023)

Responsible and Trusted AI: An Interdisciplinary Perspective

Responsible and Trusted AI: An Interdisciplinary Perspective (2025)

Sophie Kerstan[1(✉)], Kevin Baum[2,3], Thorsten Helfer[3,4], Markus Langer[1,3], Eva Schmidt[5,6], Andreas Sesing-Wagenpfeil[7], and Timo Speith[3,8]

[1] Department of Psychology, University of Freiburg, 79085 Freiburg Im Breisgau, Germany
sophie.kerstan@psychologie.uni-freiburg.de
[2] German Research Center for Artificial Intelligence (DFKI), Saarland Informatics Campus, 66123 Saarbrücken, Germany
[3] Algoright e.V., 66121 Saarbrücken, Germany
[4] CISPA Helmholtz Center for Information Security, 66123 Saarbrücken, Germany
[5] Department of Philosophy and Political Science, TU Dortmund, 44227 Dortmund, Germany
[6] Lamarr Institute for Machine Learning and Artificial Intelligence, 44227 Dortmund, Germany
[7] Institute for Legal Informatics, Saarland University, 66123 Saarbrücken, Germany
[8] Department of Philosophy, University of Bayreuth, 95448 Bayreuth, Germany

Abstract. As Artificial Intelligence (AI) continues to shape individual lives, institutional processes, and societal structures, ensuring its responsible and trusted development has become a critical imperative. However, meeting this imperative is far from straightforward. AI systems frequently lack transparency and are embedded in environments where the distribution of responsibility and accountability is unclear, normative standards are disputed, and system behavior is unpredictable. The Responsible and Trusted AI track at AISoLA 2025 addresses these and similar challenges by fostering interdisciplinary collaboration across philosophy, law, psychology, economics, sociology, political science, and informatics. This introduction outlines the motivation for the track, emphasizing the sociotechnical embeddedness of AI and the need for approaches that go beyond technical performance to consider questions related to trust and responsibility. It highlights three core themes explored in this year's contributions: democratic legitimation and normative alignment, legal compliance and human oversight, and runtime safety in high-risk contexts. Together, these contributions underscore the importance of interdisciplinary discussions to navigate normative ambiguity, regulatory uncertainty, and behavioral unpredictability in AI systems. The track aims to advance dialogue and collaboration that support the development and deployment of AI systems that are not only effective but are also designed and implemented responsibly and can be trusted.

1 Introduction

Artificial Intelligence (AI) systems are becoming increasingly integrated into everyday life. They support decision-making in healthcare, influence access to financial services, shape how public infrastructure is managed, and affect what we consume and learn. In short, AI plays a structuring role in how individuals, institutions, and societies operate.

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 141–145, 2026.
https://doi.org/10.1007/978-3-032-07132-3_9

This widespread integration brings considerable opportunities. AI can support scientific discovery, help detect diseases earlier, tailor services to individual needs and enhance productivity. However, it also introduces complex challenges and risks. These include, among others, a lack of transparency in automated decisions, blurred lines of accountability, unclear avenues for meaningful oversight, and the risk of reinforcing social inequalities. Increasingly, AI systems act in ways that are hard to explain, challenging to audit, and difficult to contest.

To harness the opportunities of AI while managing its challenges and risks, we must ensure that their development, deployment and use is guided by ethical reflection, legal scrutiny, and social awareness. The concept of responsible and trusted AI encapsulates these requirements. It encompasses a wide range of topics and issues spanning ethical reflection on the values that shape AI behavior, mechanisms for meaningful human oversight, technical assurances around safety and reliability, and institutional frameworks capable of fostering accountability and public trust. This means that, in the context of AI, responsibility and trust are not just attributes of technological systems. Rather, they are social and societal processes that require alignment between technical capabilities, normative expectations, and behavioral and regulatory realities. Tackling AI's challenges and mitigating its risks thus requires more than technical ingenuity—it demands a genuinely interdisciplinary effort. For instance, a system's ability to operate reliably under uncertain conditions is closely tied to how its behavior is interpreted, how oversight is structured, and how legitimacy is established in the face of competing values.

Following the successful introduction of similar tracks at AISoLA 2023 and 2024, the *Responsible and Trusted AI* track at AISoLA 2025 is grounded in this interdisciplinary imperative. It brings together scholars from philosophy, law, psychology, economics, sociology, political science, and informatics to address ethical, societal, and governance-related questions raised by the development, implementation, and regulation of AI systems.

In this introduction, we first expand on the motivation for this year's track and the need for interdisciplinary discourse in approaching responsible and trusted AI. We then introduce three papers that address three complementary core issues of this theme: AI alignment and democratic legitimation, human oversight and regulatory compliance, and runtime safety monitoring in high-risk contexts.

2 The Imperative for Responsible and Trusted AI

The increasing relevance of AI systems across diverse sectors prompts foundational questions: What objectives should these systems pursue? Who decides what is appropriate or fair? How can we ensure that AI behavior is not only efficient, but justifiable? And how do we maintain humans' ability to oversee AI systems, even when these systems operate largely autonomously and in ways that are difficult to understand and anticipate?

These questions arise because AI systems do not operate in a vacuum. They interact with, and are shaped by, the normative, social, and legal contexts in which they operate. At the same time, they influence how decisions are framed, which options are made available or prioritized, and how outcomes are distributed. In doing so, they often encode—explicitly or implicitly—assumptions about what is relevant, fair, or desirable.

Considering this sociotechnical embeddedness, responsibility means ensuring that AI systems not only perform tasks effectively but also do so in ways that are ethically defensible, attuned to the social contexts in which they operate, and legally compliant. Trust, in turn, is often invoked as a requirement for successful AI adoption. Yet trust is neither inherently positive nor always justified. It can be misplaced, leading to the acceptance of outcomes that ought to be challenged. To mitigate this risk and design and implement AI in ways that make it worthy of trust, we must examine and better understand the conditions under which trust arises, why it is granted and how we can ensure that it is granted justifiably.

Responsible and trusted AI thus requires attention not only to performance metrics or system reliability, but also to issues of justification, accountability, user understanding, and empowerment. Accordingly, responsible and trusted AI must be approached as a *sociotechnical* phenomenon that transcends disciplinary boundaries.

3 The Need for Interdisciplinary Approaches

As previously exemplified, addressing AI-related challenges and risks cannot be done within the bounds of a single discipline. Each field brings critical insights that are indispensable but incomplete on their own. Philosophy helps articulate the moral and conceptual frameworks that guide alignment and legitimacy. Law provides the structure for rights, obligations, and remedies. Psychology contributes knowledge on human cognition, trust formation, and interaction design. Economics brings tools to analyze incentives, resource allocation, and decision environments. Sociology sheds light on the societal dynamics, institutional norms, and structural inequalities that shape and are shaped by AI. Political science examines legitimacy, democratic participation, and governance structures. Informatics and engineering supply the technical means to implement and evaluate AI systems.

Importantly, many challenges in responsible and trusted AI exist at the boundaries of these disciplines. For example, the question of whether a user is coerced by an AI system depends on philosophical definitions and psychological evidence. Similarly, the effectiveness of human oversight mechanisms hinges not just on their formal presence, but on their legal enforceability, and organizational context.

Interdisciplinary collaboration is therefore essential not only to combine knowledge, but to clarify assumptions, identify blind spots, and align goals across disciplines. This track encourages precisely these kinds of exchanges.

4 Core Interdisciplinary Themes in This Year's Track

A unifying thread running through this year's contributions is the challenge of navigating ambiguity and uncertainty in the sociotechnical embedding of AI systems. As AI technologies increasingly interact with humans and operate in complex and high-stakes environments, the question is not merely whether these systems function as intended, but how normative, legal, and empirical standards can constrain their behavior, influence their implementation, and guide their design. The three papers approach this question from different disciplinary perspectives but share a focus on situations where questions around human freedom, oversight, and safety arise due to ambiguity and uncertainty in normative standards, regulatory frameworks, and AI behavior.

One dimension of this problem space concerns the justification of normative constraints in AI behavior. In their contribution, Steingrüber and Baum critically examine democratic approaches to AI alignment that seek to legitimize normative constraints by grounding them in affected stakeholders' preferences rather than expert judgment [1]. They systematically analyze both instrumental and non-instrumental justifications for democratic alignment, focusing particularly on the argument that democratic processes can prevent illegitimate coercion through AI systems. Their analysis reveals that the coercion-prevention justification faces significant challenges: whether AI-imposed constraints actually undermining user freedom depends not only on how those constraints were determined, but crucially on background conditions such as the availability of alternative systems and the practical burdens users face in accessing them. By exposing the deep normative and epistemic uncertainties involved in defining what AI systems should do, their work highlights the fundamental difficulty of legitimizing AI behavior without relying on contestable theoretical assumptions, ultimately suggesting that hybrid approaches combining expert knowledge with democratic input may be necessary.

Another focal point is the operationalization of legal and institutional requirements for human oversight. In their paper, Langer, Lazar, and Baum explore how oversight obligations under the EU AI Act can be meaningfully tested [2]. They argue that checklist-based assessments risk superficial compliance and fail to capture the real-world complexity of human-AI interaction. Instead, they propose hybrid approaches that combine standardization with empirical evaluation, grounded in psychology and human-computer interaction. Their work shows how ambiguity in legal language and variance across application contexts generate regulatory uncertainty. They stress that effective oversight is dependent on interdisciplinary insight into human behavior, organizational constraints, and system design.

The final contribution turns to technical assurances under behavioral uncertainty, specifically in the domain of autonomous vehicles. Ehlers and colleagues develop a runtime monitoring technique that uses activation pattern analysis and statistical guarantees to detect when a perception system operates outside its training domain [3]. Rooted in the ISO 21448 SOTIF standard, their method provides interpretable and narrowly scoped safeguards for AI behavior in open-ended, safety-critical contexts. Their work addresses an aspect of responsible AI that is distinct but closely related to those addressed by the previously outlined contributions. Specifically, it considers how to build confidence in system behavior when full formal specification is impossible, and how such mechanisms can support safety, oversight, and post-deployment trust.

Together, these contributions illustrate that responsible and trusted AI depends on interdisciplinary strategies to cope with ambiguity and uncertainty. Indirectly, they also reflect a shared understanding that trust in AI systems should be ensured not only through technical robustness, but through transparency in how values are embedded, how legal compliance is ensured, and how AI system behavior is monitored over time.

5 Conclusion

This year's Responsible and Trusted AI track contributes to a broader understanding of how AI systems can be aligned with ethical and democratic values, embedded within legal structures, and equipped with safeguards that account for uncertainty and risk. The three papers in this chapter each exemplify this integration. Steingrüber and Baum provide a normative lens on alignment and coercion [1]. Langer et al. explore how to test regulatory compliance with human oversight requirements [2]. Ehlers et al. offer a technical method for ensuring runtime safety [3].

By drawing on philosophy, law, psychology, economics, sociology, political science, and informatics, the contributions move beyond disciplinary silos and toward an integrated understanding of responsible and trusted AI. This interdisciplinary nature is central to the track's value.

As we prepare for the in-person discussions at AISoLA 2025, we look forward to engaging with these contributions and the perspectives they elicit. We hope that the work presented in this track will spark critical debate, foster new interdisciplinary collaborations, and contribute meaningfully to the ongoing effort to shape AI systems that are reliable and worthy of trust in meaningful ways.

References

1. Steingrüber, A., Baum, K.: Justifications for Democratizing AI Alignment and Their Prospects (in this volume)
2. Langer, M., Lazar, V., Baum, K.: On the Complexities of Testing for Compliance with Human Oversight Requirements in AI Regulation (in this volume)
3. Ehlers, R., Kamdoum Deameni, L., Maslov, N.: Supporting a SOTIF Safety Argument by Activation Pattern Monitoring with Statistical Guarantees (in this volume)

Justifications for Democratizing AI Alignment and Their Prospects

Andre Steingrüber(✉) and Kevin Baum

German Research Center for Artificial Intelligence (DFKI), 66123 Saarbücken, Germany
{andre.steingrueber,kevin.baum}@dfki.de

Abstract. The AI alignment problem comprises both technical and normative dimensions. While technical solutions focus on implementing normative constraints in AI systems, the normative problem concerns determining what these constraints should be. This paper examines justifications for democratic approaches to the normative problem—where affected stakeholders determine AI alignment—as opposed to epistocratic approaches that defer to normative experts. We analyze both instrumental justifications (democratic approaches produce better outcomes) and non-instrumental justifications (democratic approaches prevent illegitimate authority or coercion). We argue that normative and metanormative uncertainty create a justificatory gap that democratic approaches aim to fill through political rather than theoretical justification. However, we identify significant challenges for democratic approaches, particularly regarding the prevention of illegitimate coercion through AI alignment. Our analysis suggests that neither purely epistocratic nor purely democratic approaches may be sufficient on their own, pointing toward hybrid frameworks that combine expert judgment with participatory input alongside institutional safeguards against AI monopolization.

Keywords: AI Alignment · Legitimacy · Democratic Justification · Public Reason · Value Imposition

1 Democratic Approaches to the Normative Problem of AI Alignment

The AI alignment problem consists of two sub-problems: a technical problem and a normative problem [9, p. 412-13]. The technical problem is a question of machine ethics and requires us to find algorithmic implementations of normative constraints that effectively regulate the behaviour of AI systems. The normative, primarily philosophical problem, on the other hand, requires us to determine what these constraints should be. In this paper we will be concerned with the normative problem and potential justifications for solving it democratically.

There are two main ways to determine the content of an AI's normative constraints. The first is to take a top-down approach and determine the content

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 146–159, 2026.
https://doi.org/10.1007/978-3-032-07132-3_10

of normative constraints by consulting normative philosophical theories or by deferring to people identified as normative experts, see [1,3,4,14,19]. Secondly, one can take a bottom-up approach, letting relevant stakeholders—for instance, all those affected by an AI's alignment (directly or indirectly)—determine the content of its normative constraints, see [2,7,10,13,22]. Call the former "epistocratic approaches" and the latter "democratic approaches".[1]

It is important to note, however, that democratic approaches may be democratic in name only, as they may fail to be genuinely democratic depending on the procedure being used to determine the normative constraints from the input of the people. Some procedures, if done correctly, are apt to be democratic—e.g., voting, sortition, or deliberation—others, like a knockout tournament in bowling or a debating contest, much less so.

Schuster and Kilov [21] argue that current proposals for democratic approaches all invoke procedures that fail to be democratic. However, we believe this is based on a misunderstanding. Let us clarify, because this misunderstanding comes up often in the alignment literature: The approaches that Schuster and Kilov discuss are crowdsourcing normative judgement, reinforcement learning from human feedback (RLHF), and constitutional AI. Yet, these three techniques should *not* be understood as solutions to the normative problem, but rather as solutions to the *technical* problem.[2] In the case of crowdsourcing and RLHF, normative constraints are given implicitly in the form of a large number of individual human normative judgements [7,13], and with constitutional AI, normative constraints are given explicitly in the form of a list of principles formulated in natural language [4]. The primary aim of these techniques is to implement normative constraints in AI systems and not to determine what the normative constraints should be.

Crowdsourcing, RLHF and constitutional AI are all compatible both with epistocratic and democratic approaches to the normative problem. Although constitutional AI may sound like it particularly lends itself to epistocratic approaches, and crowdsourcing and RLHF like they are especially suited for

[1] Pluralistic approaches would combine epistocratic and democratic elements to determine an AI's normative constraints. As we will explain in the next section, the task of producing normative constraints for an AI can be broken up into three steps: For every scenario, we need to (i) identify the relevant reasons, (ii) measure the relative strength of these reasons, and (iii) aggregate the relevant reasons to form overall deontic verdicts that formulate the AI's normative constraints. Approaching these three steps in a pluralistic fashion, one can partition the set of scenarios and handle one subset epistocratically and the other subset democratically, or one can let epistocratic approaches take care of certain steps of the procedure and let democratic approaches do the remaining steps; or, alternatively, one can involve both epistocratic and democratic approaches in a single step, e.g., identifying the relevant reasons by eliciting people's judgements on the matter and then letting experts add missing reasons they consider important, or letting them veto against particular reasons.

[2] See, e.g., also [9, p. 414], [12, pp. 12-13], and [15, p. 2672], for instances where RLHF and constitutional AI are treated as solutions to the normative problem.

democratic approaches, there is no such association for any of these techniques. In general, it is possible to determine the list of principles necessary for constitutional AI both via an epistocratic or a democratic approach; either we consult normative theories to derive a constitution, or we ask all affected people to determine one. Likewise for RLHF and crowdsourcing, what the input data should be can either be decided by normative experts, or by the affected public. We must be careful to distinguish between the procedures invoked to solve the normative problem and techniques used to tackle the technical problem.

Having made this clarification, let's return to the two approaches to the normative problem. Some authors argue that democratic approaches, if they are actually democratic, should be favoured over epistocratic ones because they allow us to avoid putatively morally undesirable aspects of epistocratic approaches [10,12,13]. Here are some moral reasons that are claimed to disfavour epistocratic approaches:

> 'The lack of a broad, inclusive, and democratic process for determining these values can lead to AI systems that disproportionately reflect the interests of specific groups, exacerbating existing inequalities and failing to serve the broader public good.' [12, p. 11]

> '[W]e follow a bottom-up approach to Delphi for an important ethical concern: [...] implementing the top-down approach would force scientists to impose their own value choices and principles in the system they build, which is not an appropriate social role for scientists alone.' [13, p. 7]

> '[E]fforts to align AI systems with a given moral schema may lead to unjust value imposition or even domination.' [10, p. 3]

We can group the putative reasons speaking against epistocratic approaches into two categories: Instrumental reasons against epistocratic approaches (and for democratic approaches), and non-instrumental reasons against epistocratic approaches (and for democratic approaches).[3] If epistocratic approaches were to "exacerbate inequalities" or "fail to serve the public good", they would be instrumentally worse than democratic approaches, because adopting them would have morally worse consequences. If, on the other hand, pursuing epistocratic approaches were to constitute "value imposition" or "domination", they would be non-instrumentally worse, because they are inherently, i.e., independently of their consequences, morally objectionable.

Proponents of democratic approaches are not always explicit about what kind of justification it is that speaks in favour of their theory, and they don't always consider what theoretical resources epistocratic approaches can draw on that may undercut the justifications that are supposed to support democratic approaches. Therefore, in this paper, we want to unpack the reasons that may be

[3] Compare [8] for this taxonomy of justifications for democratic practices.

used to justify democratic approaches and estimate how promising they are. We want to suggest to the proponents of democratic approaches the most promising justificatory avenues, but also point out what questions they have to answer to pave those paths. Mainly, we will focus on non-instrumental reasons, but we will also briefly touch upon the instrumental ones. Two possibilities for what might be inherently bad about epistocratic approaches will be discussed: (i) They give some people illegitimate authority over other people. (ii) Through them some people will be illegitimately coerced by others. We will argue that the latter is the more promising argumentative route for proponents of democratic approaches. However, whether it succeeds in justifying democratic approaches over epistocratic approaches depends on at least four things: that users of an AI can really be coerced through the AI's alignment; that, if users of an AI can really be coerced through the AI's alignment, this would be illegitimate; that democratic approaches can produce a democratic justification that would justify the coercion and thereby prevent illegitimate coercion; and that epistocratic approaches cannot prevent the illegitimate coercion.

However, before we turn to discuss instrumental justifications and subsequently non-instrumental justifications, we first want to consider a crucial motivation and enabling condition for democratic approaches: reasonable normative disagreement.

2 Normative Disagreement Leaves a Justificatory Gap

The observation that reasonable people can deeply disagree when it comes to normative matters is one motivation for proponents of the democratic approach to pursue their project [2,10,12,13]. It is worthwhile to consider how exactly that is so, to get clearer on what the aims and obstacles of democratic approaches are. The short version is this: The empirical fact of normative (and metanormative) disagreement makes us normatively (and metanormatively) uncertain, i.e., we are unsure what the right thing to do is (and whether there even is a uniquely right thing to do). This uncertainty eliminates what would be a straightforward justification for any potentially illegitimate state of affairs. If we were normatively (and metanormatively) certain, we could simply show that the normative constraints we are implementing are (objectively) correct. Since such a theoretical justification is unavailable, this makes it possible that, by means of an AI's normative constraints, people are given *illegitimate* authority over other people, or that some people are being *illegitimately* coerced by others. Let us consider this in more detail.

We are facing the normative problem under both normative and metanormative uncertainty. That is, neither are we certain what the normative ground truth is, nor are we certain whether there even is a normative ground truth and how to find out about it.

We are normatively uncertain because we can observe that reasonable people can widely diverge in their judgements about what reasons are relevant for a decision and how their strength compares to each other, and because different normative theories, like theories of normative ethics, can have very different answers to these questions [17]. That is, our normative uncertainty is the rational response to observed intersubjective and intertheoretical normative disagreement.

Likewise for metanormative disagreement. It is the rational response to observed intersubjective and intertheoretic metanormative disagreement. We are metanormatively uncertain in at least three respects: We are uncertain whether there is a normative ground truth, i.e., whether there are robustly mind-independent normative reasons. We are uncertain whether this normative ground truth is unique, i.e., whether normative reasons hold absolutely or only relative to some frame of reference. And we are uncertain whether and how we can have knowledge about this normative ground truth, i.e., whether there is a reliable method to identify, measure and aggregate normative reasons.

A short digression: We are deliberately speaking about normative and metanormative uncertainty in general and not just about moral and metaethical uncertainty in particular, because an AI's normative constraints are not exhausted by moral constraints. We don't just want to know what is morally permissible, impermissible or obligatory to do for an AI system, we want to know what is overall permissible, impermissible or obligatory [5]. To properly align AI systems they have to be sensitive to normative domains other than the moral domain. Consider, e.g., that some things that are morally permissible are not legally permissible, like taking food from the supermarket's bin, or they are not socially permissible, like talking much too loud in public spaces. To know what we and what an AI should do—Are we allowed to stand in the middle of the escalator blocking other people from walking past us?—we have to consider all relevant reasons from different relevant normative domains and weigh them against each other in order to arrive at an all-things-considered *overall* deontic verdict and not just an all-things-considered *moral* deontic verdict. To solve the normative problem, we thus have to: (i) *identify* which practical reasons from which normative domain are relevant for a decision, (ii) *measure* the strength of the relevant reasons, and (iii) *aggregate* the relevant reasons according to their strength to form an all-things-considered overall reason that grounds an overall deontic verdict.

The fact that we are seeking overall reasons that play the role of overall normative constraints exacerbates our normative and metanormative uncertainty. For one, if we consider non-moral normative domains in isolation there may be even less common ground in people's judgements or conversely even more normative disagreement. Just consider the diverse social norms or legal norms that people take to hold. Which of them should we choose to align AI with? But what's more, since the reasons from different normative domains interact, this introduces an entirely new dimension of normative/metanormative uncertainty. How exactly do moral reasons, reasons of politeness, and legal reasons interact, for example? Some may tend to let legal and politeness reasons be able to take precedence over moral reasons, others will think that moral reasons always override reasons from other domains. All these uncertainties can accrue and reflect in our uncertainty about the all-things-considered overall normative reasons that, in the end, are supposed to figure as normative constraints for an AI system.

Now, how exactly do normative and metanormative uncertainty motivate democratic approaches? They do so insofar as they are necessary conditions for the possibility of the illegitimacy of authority or coercion. If we were certain that some objective all-things-considered overall reason holds, then this would give us a justification to do as the reason demands. If we had decisive evidence (whatever that would look like) for the truth of a certain practical normative judgement—One ought to φ—then we would have all-things-considered theoretical reason to believe that one ought to φ which in turn would constitute a contributory practical reason to φ. Normative certainty would therefore put us in a position to justify and thereby legitimise authority or coercion; we would be able to show that some demands are not discretionary but well founded. Conversely, this is how normative and metanormative uncertainty is a necessary condition for unjustified authority or coercion: it eliminates a sure theoretical justification that could always legitimise potentially illegitimate authority or coercion; when we don't know what the normative ground truth is, or we don't know how to find out what it is, or are not even sure that there is one, then we can't appeal to it to safely justify a potentially illegitimate state of affairs.

Normative and metanormative uncertainty thus leave us with a justificatory gap, one that democratic approaches are motivated to fill. The democratic aim is to compensate for the *missing theoretical* justification of an AI's normative constraints with a *political* justification. The idea being, if all people affected by an AI's normative constraints get to have a say in what these constraints are, this legitimises any potentially illegitimate authority or coercion by means of an AI's normative constraints. Preventing illegitimate authority or coercion is supposed to non-instrumentally justify democratic solutions to the normative problem. Before we consider non-instrumental justifications, however, let us briefly say a few words about instrumental justifications for democratic approaches.

3 Instrumental Justifications for Democratic Alignment

Proponents of democratic approaches may justify their preferred solution to the normative problem by arguing that it, in some sense, works better than epistocratic approaches; employing democratic approaches has better consequences than not doing so. We want to mention two ways in which this might be the case, and on which defenders of the democratic approach could focus.[4]

First, one may try to argue that if we let the people that are going to be affected by the behaviour of an aligned AI decide how it ought to be aligned, then the aligned behaviour of the AI will be better *for* the people. The idea is that people know best what is good for them, or at least better than normative experts and their theories. Thus, if we let them decide, instead of only the experts, they will be better off than they otherwise would have been.

[4] For a general description of both of them, see [8].

But there is still quite some argumentative work left to be done for this justification to really get off the ground. First of all, proponents of the democratic approach need to decide whether they want to read the counterfactual "If people have a say in what the normative constraints of an AI are, they would be better off (with respect to the AI's behaviour towards them) than they otherwise would have been" generically or specifically. Do they aim for a general justification of democratic approaches and want to roughly say "Typically, if people have a say in what the normative constraints of an AI are, they would be better off (with respect to the AI's behaviour towards them) than they otherwise would have been"? Or do they aim for a case-by-case justification and want to say "In this case, if people have a say in what the normative constraints of an AI are, they would be better off (with respect to the AI's behaviour towards them) than they otherwise would have been"? The latter justification is weaker but also comparatively easier to come by.

Under both readings, democratic approaches still have to argue that people would *actually* be better off than they otherwise would have been. It does not seem implausible to suppose, e.g., that normative experts are subject to biases that reflect in their normative verdicts and that consequently would disadvantage certain groups of persons. If these people get to have a say, then, most likely, they will not disadvantage themselves, i.e., plausibly they would be better off. However, proponents of democratic approaches should be careful to take epistocratic approaches seriously and not to argue against straw men of them. It might be a real risk that epistocratic approaches arrive at normative constraints that are biased, but to criticise epistocratic approaches this risk has to be estimated, and additionally, democratic approaches need to show that they do not run this risk. Further, to argue successfully that it is better for people if they can democratically participate, proponents of democratic approaches have to react to objections that invoke cases where people seem to vote against their best interest; think Brexit, Trump, the climate crisis, etc.

Another possible instrumental justification we want to mention relates to the idea of the wisdom of crowds. The claim would be that, although epistocratic approaches consult the judgement of normative experts, democratic approaches are better at producing more correct results. This is an epistemic justification because the point is supposed to be that (under certain assumptions) democratic processes are better at tracking the normative facts. Typically, Condorcet's Jury Theorem is being used to argue for this point. Roughly, it states that the probability that a majority of voters choose the correct option approaches 1 as the number of voters increases. That is, the bigger the electorate, the more reliable the result of their vote [11].

However, Condorcet's Jury Theorem relies on unrealistic assumptions. For it to hold, one needs to assume that voters' judgements are probabilistically independent of each other, and that voters are generally competent, evidenced by the fact that they are more likely to vote for the correct option than for the incorrect option. In real-life cases these assumptions are almost never satisfied. The assumptions can, however, be weakened to make the jury theorem applicable for real cases [11]. Even then though proponents of the democratic approach have to show that the weakened assumptions hold in the case of AI alignment they are considering. And they need to respond to objections, two of which we want to allude to. First, democrats have to make sure that epistocrats cannot also make use of the jury theorem, with the difference being that only normative experts comprise the electorate. And second, to employ the jury theorem one has to assume that there is an objective fact about the matter that is being voted on. In the present context of the normative problem the matter would be normative, and to assume that there is an objective fact about these matters would be a metanormative assumption. Such an assumption might be in tension with the assumption of metanormative uncertainty democratic approaches are motivated by.

Proponents of democratic approaches can argue for their proposed solution to the normative problem by resorting to these and other instrumental justifications. To reap the justificatory fruits they have to show that the advertised consequences—prudentially or epistemically better decisions—are actually achievable in the case of AI alignment, and they have to show that epistocratic approaches do not have access to the same benefits in different ways. Another way to justify democratic approaches is through non-instrumental justifications. We will turn to them now.

4 Non-instrumental Justifications for Democratic Alignment

Above, we have quoted Gabriel and Keeling who worry that epistocratic approaches may lead to illegitimate "value imposition or even domination" [10, p. 3]. This exemplifies a non-instrumental objection to epistocratic approaches. More detailed, the worry is that, through an AI's alignment, people can be indirectly subjected to normative standards they do not subscribe to themselves. Since epistocratic approaches cannot close the justificatory gap left by normative and metanormative uncertainty, they cannot justify such subjection which makes it illegitimate. For example, if your personal AI assistant does not let you buy meat because that would be against its normative constraints, you are being subjected to normative standards to which you do not subscribe. Or, if a generative AI is uncompliant with your request to gender an email draft because its alignment forbids it to do so, other people's values are being imposed on you.

But what exactly do we mean by "value imposition", "domination", and "subjection"? Two possible interpretations are that they either refer to authority over the users of AI, or to the coercion of users of AI. A person with (justified) authority can issue commands or make claims that generate real reasons for action for other people [18]. For example, within certain confines, teachers are typically taken to have (justified) authority over their pupils. And a person equipped with coercive power can restrict other people's freedom to act as they desire. For example, within certain confines, policemen are typically taken to wield coercive power.

What is at issue in the case of contentious AI alignment? Arguably, it is coercive power rather than authority. The question of authority would only arise if an AI system were to be deployed in a way where it issues commands or makes claims on people. The question of coercive power, on the other hand, arises as soon as an AI system restricts people's freedom to act as they desire. This can happen rather quickly. If a self-driving vehicle does not let you drive above a certain speed limit because of its normative constraints, then it doesn't command you to drive slower, it simply makes you so. And if a large language model does not let you write your text in gender-sensitive language, it makes no claim on you to not do so, it just doesn't use gender-sensitive language. We could multiply examples but the point is: Both the question of an AI's authority and its coercive power can be pertinent but we take the threat of coercion to be the more pressing and focus on it in the following.[5]

Proponents of democratic approaches have to argue for four things in order to be able to claim that preventing illegitimate coercion non-instrumentally justifies democratic approaches as compared to epistocratic approaches: They have to argue (i) that it is indeed possible for people to be coerced through an AI's normative constraints, (ii) that such coercion, if it is possible, would be unjustified, (iii) that democratic approaches can produce a democratic justification that would justify the coercion and thereby prevent illegitimate coercion, and (iv) that epistocratic approaches cannot prevent the illegitimate coercion. Let us make a few remarks concerning each proposition.

Is it possible for the users of an AI to be coerced through the AI's normative constraints? Two quick clarifications to begin with: First, if an AI's alignment is coercive, the primary coercer is not the AI itself but the person or organisation that defines the normative constraints. The AI is only the means of (potential) coercion. A bit more verbosely we are asking: Do the people who define the normative constraints of an AI coerce the users of the AI if (some of) the chosen normative constraints are not endorsed by the users? Second, and following from this, if users are being coerced, then only indirectly so. The primary (potential) coercers are the people who decide how the AI is aligned, but they don't compel the users directly; rather, their coercion is mediated by the AI. That an AI's alignment can only be indirectly coercive does not speak against it really being coercive. If you can be coerced by having someone limit what you can do with your bank account—think, abusive relationship—you can also be coerced by having someone limit what you can do with an AI.

More needs to be said to convincingly argue that users of an AI can be indirectly coerced by its normative constraints. Let's assume that an argument can be given for that. What would have thereby been shown is that it is *in principle possible* for an AI's alignment to be coercive. What has not been shown is that there is an *actual* case where it *actually* is coercive. For an AI's alignment to be *actually* coercive would not just depend on the relationship between the

[5] See also Ripstein [20], who argues that, in general, the function of democratic justification is to legitimise coercion rather than authority.

users and the aligned AI but crucially also on certain background conditions.[6] We said that a person equipped with coercive power can restrict other people's freedom to act as they desire. Conversely this means if a person is free to act as she pleases then she is not being coerced. This matters for the case of normatively coercive AI in the following way: Say, for normative reasons, your AI assistant is noncompliant with your request to buy meat. Are you being coerced by this? Well, it depends on whether you are still free to buy meat, maybe with the help of another, differently aligned AI, or simply on your own. You wouldn't be free to do so, for example, if using the specific AI assistant were the *de facto* or *de jure* standard for going shopping. You would then not be able to buy meat, at least not without great opportunity cost. But if that is not the case, and without much ado you can just go and use another AI or buy the meat yourself, then you are not being coerced by the normatively noncompliant AI. You are no more being coerced than you would be when you can only buy vegan products at your local supermarket, or when you have to wear "gender appropriate clothes"—no skirts for men, no trousers for women, etc.—at your bowling club. The users of the AI, the supermarket, or the bowling club may be compelled to use them in a certain way, but this is not coercion as long as they can freely go elsewhere to use another AI, another supermarket, another bowling club.

The next point proponents of democratic approaches would have to argue for, in order to strengthen the non-instrumental justification in favour of their solution, is that coercion by means of an AI's alignment, if it is possible, would be unjustified. Two important ingredients for such an argument would presumably be the prima facie wrongness of coercion, and the fact that we are normatively and metanormatively uncertain. The first ingredient, the prima facie wrongness of coercion, may be used to establish that prima facie coercion stands in need of justification, much like murder or marital infidelity would. And the second ingredient, normative and metanormative uncertainty, may be used to establish that no straightforward theoretical justification for the AI's normative constraints is available, and consequently also not for the coercion by means of them. Thus, other things being equal, coercion by means of an AI's alignment would be unjustified.

Observe however, that there is a tension between the two ingredients. If we are normatively uncertain, how can we purchase the assumption that coercion is prima facie wrong?[7] We are deliberately only talking about a tension and not a contradiction because proponents of democratic approaches may be able to argue that the two ingredients are consistent with each other. For example, because our normative uncertainty is not evenly distributed over all normative propositions; about some we are more certain, about some we are less certain. Coercion being prima facie wrong perhaps is of the first kind, while overall we are still normatively uncertain. Whether in this way or differently, if proponents

[6] In a similar context Kolodny [16, pp. 97-101] calls these background conditions "tempering factors".

[7] We thank two reviewers for raising that point.

of the democratic approach buy into both ingredients then they have to address the apparent tension between them.

Another point that needs to be addressed is if there are other justifications for (the choice of) certain normative constraints, apart from a sure theoretical justification, or the democratic justification that democratic approaches aim for. If there is another possible justification, then coercion through AI alignment would be justifiable and proponents of democratic approaches would lose the non-instrumental reason in favour of their solution. Candidates for such a justification are decision rules that are explicitly designed to deal with normative uncertainty. For example, MacAskill, Bykvist and Ord [17] defend a rule called "maximise expected choiceworthiness". Analogous to descriptive uncertainty they treat normative disagreement as data to approximate the correct normative constraints by assigning weights to different normative hypotheses, where different normative hypotheses determine the choiceworthiness of an outcome. Maximise expected choiceworthiness then demands to choose the action that leads to the outcome with the highest sum of weighted choiceworthiness. Although we cannot have a sure theoretical justification for any particular normative constraint, we may be able to have a practical justification to choose certain constraints over others by means of decision rules like this that deal with normative uncertainty. The practical justification we get from any such rules inherits its strength from the strength of the theoretical justification for the particular decision rule, meaning that proponents of democratic approaches either have to critique the theoretical justification for the rule, or they have to argue that the practical justification for choosing certain normative constraints we get from rules like maximise expected choiceworthiness is in general of the wrong kind.

The next point proponents of democratic approaches would have to argue for, in order to strengthen the coercion-based non-instrumental justification in favour of their solution, is that they can actually produce a democratic justification that is suited to justify potential coercion. A number of objections can be levelled against this, and would therefore have to be addressed by proponents of the democratic approach. Let us mention just two.

Any democratic approach will have to stipulate what the rules of their democratic game should be. Do people vote, if so, what's the voting procedure? Do people deliberate, if so, what are the rules of discourse? And so on. No matter what the rules end up being, for them to be recognisably democratic, they have to make normative assumptions.[8] For example, a vote is free and equal, or a deliberation is inclusive and non-coercive. The question however is, what justifies these normative assumptions? Some of the people affected by an aligned AI may be able to reasonably reject them. But if the democratic process is not properly justified, its output will also not be. This, like all the other points, is not a knockdown argument, rather, it is intended to raise an issue that proponents of democratic approaches have to somehow address—namely, the bootstrapping

[8] For a detailed discussion of this point in context of AI ethics in general, see [6].

problem of how to justify the very democratic procedures that are supposed to provide justification.

The same applies for this second objection. Given the deep normative disagreement between people, one may worry that whatever all people affected by an AI's alignment can agree on will only be the 'lowest common normative denominator', and much too little to really prevent the threat of illegitimate coercion. The objection here is not that no democratic justification is achieved through the democratic approach, it is that the justification is too minimal to do the job it is supposed to do. The minimal output from democratic approaches has to be beefed up, in order for AI systems to really be effectively regulated, but then the threat of illegitimate coercion re-enters again.

The last point proponents of democratic approaches would have to argue for, in order to strengthen the coercion-based non-instrumental justification in favour of their solution, as opposed to the epistocratic solution, is that epistocratic approaches do not have the resources to confront the threat of illegitimate coercion through an AI's alignment. By now we have seen that it is not so clear that this is the case. Let us mention again just two reasons for thinking that epistocratic approaches can get a handle on the problem of illegitimate coercion.

We have said that the possibility of coercion depends on certain background conditions. For example, if the use of some AI system is the *de facto* or *de jure* standard, then people may not be free to do what they desire to do without using the AI, and then they are potentially being coerced by the AI's alignment. Conversely this means, if we make sure that for all purposes there are always multiple AIs with different alignments available, then people are free to choose the AI that does not coerce them. And in general, if we control the background conditions that are necessary for coercion, we, and proponents of epistocratic approaches in particular, can prevent the threat of illegitimate coercion by preventing the threat of coercion.

Additionally, epistocratic approaches may be able to justify their choice of normative constraints and thereby justify potential coercion by means of them. To this end, they can invoke decision rules like maximise expected choiceworthiness, that explicitly take normative uncertainty into account. If an epistocratic approach is able to appropriately justify its solution of the normative problem, and is able to transfer this justification to the coercion through an AI's alignment, then this would undercut the non-instrumental justification from coercion for democratic approaches.

5 Conclusion

In this paper, we have motivated democratic approaches to the normative question of AI alignment. We have discussed instrumental and non-instrumental ways of justifying them, focussing in particular on the non-instrumental justification from coercion. We have argued that proponents of democratic approaches have to argue for four propositions in order for this justification to be successful: (i) It is possible for people to be coerced through an AI's normative constraints. (ii) Such coercion, if it is possible, would be unjustified. (iii) Democratic approaches can produce a democratic justification that would justify the coercion and thereby prevent illegitimate coercion. (iv) Epistocratic approaches cannot prevent the illegitimate coercion.

We have argued that none of the four propositions is without problems and comes for free. In particular, we have argued that there are independent ways to prevent people from being coerced by means of an AI's normative constraints. Namely, we can control the background necessary conditions for coercion, e.g., we can prevent any one AI becoming the *de facto* or *de jure* standard for certain purposes such that people are dependent on it. If epistocratic approaches can draw on this possibility, as democratic approaches can as well, then they might be able to take the sting out of the non-instrumental justification from coercion in favour of democratic approaches.

Our analysis suggests that neither purely epistocratic nor purely democratic approaches to the normative problem may be sufficient on their own. The challenges we have identified do not eliminate the potential value of democratic participation, but rather point towards more nuanced, context-sensitive solutions. At least for some application contexts, hybrid frameworks that combine expert judgement with targeted participatory input, alongside appropriate institutional safeguards that mitigate AI monopolies and in particular AI systems that are too uniformly aligned, may therefore be the most suitable paths for addressing the normative dimensions of AI alignment. The critical question for future research is determining when and how to optimally combine epistocratic and democratic elements—specifying which aspects of the normative problem benefit from expert knowledge versus democratic input, and under what institutional conditions such hybrid approaches can succeed.

Acknowledgments. We thank two anonymous reviewers and Elijah Millgram for their feedback on versions of this paper. This work is partially funded by DFG grant 389792660 as part of TRR 248 – CPEC, see https://perspicuous-computing.science, by the German Federal Ministry of Education and Research (BMBF) as part of the project "MAC-MERLin" (Grant Agreement No. 01IW24007), and from the European Regional Development Fund (ERDF) as well as the German Federal State of Saarland within the scope of the project (To)CERTAIN.

Disclosure of Interests. The authors have no competing interests to declare.

References

1. Askell, A., et al.: A general language assistant as a laboratory for alignment (2021). arXiv:2112.00861
2. Awad, E., et al.: The moral machine experiment. Nature **563**, 59–64 (2018)
3. Bai, Y., et al.: Training a Helpful and harmless assistant with reinforcement learning from human feedback (2022). arXiv:2204.05862
4. Bai, Y., et al.: Constitutional AI: harmlessness from AI feedback (2022). arXiv:2212.08073

5. Baum, K.: Disentangling AI alignment: a structured taxonomy beyond safety and ethics. In: Steffen, B., (ed.) AISoLA 2024 Post-Proceedings. Lecture Notes in Computer Science. Springer, Cham (forthcoming). arXiv preprint arXiv:2506.06286 (2025)
6. Baum, S.D.: Social choice ethics in artificial intelligence. AI Soc. **35**(1), 165–176 (2017). https://doi.org/10.1007/s00146-017-0760-1
7. Christiano, P. F., et al.: Deep reinforcement learning from human preferences. Adv. Neural Info. Process. Syst. **30** (2017)
8. Christiano, T., Bajaj, S.: Democracy. The Stanford Encyclopedia of Philosophy (2024). https://plato.stanford.edu/archives/sum2024/entries/democracy/
9. Gabriel, I.: Artificial intelligence, values, and alignment. Mind. Mach. **2**(5), 411–437 (2020)
10. Gabriel, I., Keeling, G.: A Matter of Principle? AI alignment as the fair treatment of claims, Philosophical Studies (2025)
11. Goodin, R.E., Spiekermann, K.: An Epistemic Theory of Democracy. Oxford University Press, Oxford (2018)
12. Huang, L.T., Papyshev, G., Wong, J.K.: Democratizing AI alignment: from authoritarian to democratic AI ethics. AI Ethics **5**, 11–18 (2025)
13. Jiang, L. et al.: Can machines learn morality? The Delphi experiment (2022). arXiv:2110.07574
14. Kim, T.W., Hooker, J., Donaldson, T.: Taking principles seriously: a hybrid approach to value alignment in artificial intelligence. J. Artif. Intell. Res. **70**, 871–890 (2021)
15. Kneer, M., Viehoff, J.: The hard problem of AI alignment: value forks in moral judgment. In: FAccT '25: Proceedings of the 2025 ACM Conference on Fairness, Accountability, and Transparency, pp. 2671–2681 (2025)
16. Kolodny, N.: The Pecking Order. Harvard University Press, Cambridge, MA (2023)
17. MacAskill, W., Bykvist, K., Ord, T.: Moral Uncertainty. Oxford University Press, Oxford (2020)
18. Raz, J.: The Morality of Freedom. Oxford University Press, Oxford (1986)
19. Riesen, E., Boespflug, M.: Aligning with ideal values: a proposal for anchoring AI in moral expertise. AI Ethics **5**, 3727–3741 (2025)
20. Ripstein, A.: Philos. Public Aff. Auth. Coercion **32**(1), 2–35 (2004)
21. Schuster, N., Kilov, D.: Moral disagreement and the limits of AI value alignment: a dual challenge of epistemic justification and political legitimacy. AI Soc. (2025)
22. Ziegler, D., et al.: Fine-tuning language models from human preferences (2019) . arXiv:1909.08593

On the Complexities of Testing for Compliance with Human Oversight Requirements in AI Regulation

Markus Langer[1](✉), Veronika Lazar[2], and Kevin Baum[3]

[1] Department of Psychology, University of Freiburg, Freiburg, Germany
markus.langer@psychologie.uni-freiburg.de
[2] German Federal Office for Information Security, Bonn, Germany
[3] German Research Center for Artificial Intelligence, Responsible AI and Machine Ethics Research Group, Saarbrücken, Germany
kevin.baum@dfki.de

Abstract. Human oversight requirements are a core component of the European AI Act and in AI governance. In this paper, we highlight key challenges in testing for compliance with these requirements. A central difficulty lies in balancing simple, but potentially ineffective checklist-based approaches with resource-intensive and context-sensitive empirical testing of the effectiveness of human oversight of AI. Questions regarding when to update compliance testing, the context-dependent nature of human oversight requirements, and difficult-to-operationalize standards further complicate compliance testing. We argue that these challenges illustrate broader challenges in the future of sociotechnical AI governance, i.e. a future that shifts from ensuring "good" technological products to "good" sociotechnical systems.

Keywords: Human Oversight · Auditing · AI Act · Regulation

1 Introduction

Testing for compliance with emerging legislation regarding Artificial Intelligence (AI) such as the European AI Act will be a major task for providers and deployers of AI-based systems when these systems are used for high-risk tasks [1, 2]. Some aspects of this compliance testing will resemble traditional auditing processes for classical software systems and other technologies governed by product safety regulations [1]. For instance, verifying whether AI systems have adequate documentation, ensuring cybersecurity, testing for data protection, and evaluating the accuracy of system outputs could all be achieved by defining standards and quality thresholds. Eventually, compliance testing may also draw on established best practices, such as checklist-based approaches to assess whether implemented processes and technologies comply with standards set by norming bodies. Of course, new testing procedures will also be required to assess robustness and fairness according to various criteria, particularly with regard to the immediate safety of (semi-)autonomous, agentic systems. New testing procedures and benchmarks will emerge, and services and infrastructure will build around them.

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 160–169, 2026.
https://doi.org/10.1007/978-3-032-07132-3_11

However, the AI Act and other emerging AI regulations introduce requirements that go beyond the technical characteristics of the system and requirements that traditional compliance testing methods cannot easily address. One such key requirement is effective human oversight, as outlined in Article 14 of the AI Act [3, 4]. Various countries, including Argentina, Bahrain, Uganda, and South Africa, already enforce similar but less specific requirements for human involvement in AI-driven decision-making [3]. Effective human oversight in the sense of the AI Act includes specific sub-requirements, such as ensuring that human oversight personnel remain aware of their tendency to over-rely on outputs produced by a high-risk AI system (e.g., automation bias; [5]) and that they properly understand the relevant capacities and limitations of the high-risk AI system to adequately monitor its operation (see Article 14 AI Act).

In this paper we claim that a key challenge of testing for compliance with human oversight requirements lies in balancing simple but potentially ineffective checklist-based approaches with resource-intensive and context-sensitive empirical testing of the effectiveness of human oversight of AI. Questions regarding when to update compliance testing, the context-dependent nature of human oversight requirements, and difficult-to-operationalize standards for when oversight is truly "effective" further complicate compliance testing [6]. The intricate sociotechnical interplay of technical aspects, individual factors, and environmental conditions that determine human oversight effectiveness adds additional complexity [8]. In fact, research suggests that human oversight requirements are particularly difficult to operationalize and test (see e.g., [7]). We argue that all of this illustrates broader challenges in the future of sociotechnical AI governance, i.e. a future that shifts from ensuring "good" technological products (e.g., safe products) to "good" sociotechnical systems (e.g., safe human-AI interactions in a specific context).

2 The Possible Future of Testing for Compliance with Human Oversight Requirements

The European AI Act establishes requirements for AI systems classified as "high-risk," including those used in (critical areas of) education, public administration, hiring, credit scoring, and medicine. One such requirement is human oversight, as specified in Article 14 (see Appendix A for the full text). It states that "human oversight shall aim to prevent or minimise the risks to health, safety or fundamental rights that may emerge when a high-risk AI system is used in accordance with its intended purpose or under conditions of reasonably foreseeable misuse, in particular where such risks persist despite the application of other requirements set out in this Section." These other requirements, detailed in Articles 9–13 and Article 15 of the AI Act, cover risk management, data governance, technical documentation, record keeping, transparency, accuracy, robustness, and cybersecurity.

Some key requirements of Article 14 include that human oversight personnel should be able to understand the capacities and limitations of AI systems and correctly interpret outputs. They should remain aware of their tendency for automation bias (which, according to the AI Act, refers to overtrust in AI outputs; but see [8] for the complexities and dynamics of concepts associated with trust), decide when not to use AI outputs, and override decisions when necessary. They are also supposed to intervene or interrupt a system, for example, using a stop button or a similar mechanism to halt the system in a safe state.

Standards and norms are currently being developed to guide compliance testing for human oversight requirements, including the Trustworthiness Framework developed by CEN/CENELEC and the ISO/IEC CD 42105. Part of these standards and norms will be informed by Article 14 of the AI Act and broader international governance perspectives on human oversight. While still in development, we anticipate a future of compliance testing for human oversight along a continuum between simple, checklist-based approaches and empirical testing of the effectiveness of human oversight in specific contexts.

A checklist approach would follow the model of existing compliance testing methods [1, 7] translating Article 14 requirements into assessable items for internal or external auditors. Checklist items inspired by Article 14 might include: "The human oversight person has been made aware of their tendency to overtrust outputs of AI-based systems", "The human oversight person has received adequate training that enables them to understand the capacities and limitations of the AI-based system they oversee", or "There is a stop button that allows the human oversight person to intervene in the operation of the AI-based system". Clearly, this list is not exhaustive and the requirements would need to be refined. While such checklists could provide a straightforward compliance mechanism, they may fall short of the AI Act's broader goal of *effectively* mitigating AI-related risks [1]. Moreover, developing an exhaustive checklist will be challenging. The examples above are direct translations from Article 14. Clearly there can be an infinite number of requirements with varying degrees of specificity, for instance, requirements concerning the person who will be the human oversight person (e.g., specific skills they must possess), or work design of human oversight jobs (e.g., specific maximum durations for human oversight tasks) [9].

Empirical testing of the effectiveness of human oversight in specific contexts could address some of the limitations of checklists. This approach would require testing the actual effectiveness of human oversight in high-risk contexts and empirically demonstrating compliance with AI regulatory requirements [3]. It could involve studies where human oversight personnel monitor AI systems for a set duration, assessing whether they detect erroneous or problematic outputs, intervene in system operation when necessary, and accurately override inadequate AI-generated decisions. Another option could involve comparing different human oversight designs to determine which best meets regulatory requirements. Human oversight design, as outlined by Sterz et al. [9] is a sociotechnical design question. It encompasses technical aspects (e.g., optimizing explainability approaches to support and amplify human oversight), individual factors (e.g., selecting and training oversight personnel), and environmental conditions (e.g., job design and working conditions). For instance, a controlled experiment could test various explainability approaches to assess which most effectively supports oversight [10–12]. The main advantage of this approach is that it provides empirical evidence on the effectiveness of human oversight and how to optimize it.

However, this empirical approach demands significant resources for planning, conducting, analyzing, and interpreting studies. Empirical testing for the effectiveness of human oversight requires expertise with empirical methods and user studies. Researchers and practitioners, for instance, with a background in human-computer interaction or psychology will be required to adequately conduct empirical testing, interpreting the results, and providing recommendations on how to optimize human oversight design. Moreover, transferring insights across contexts may be challenging, as oversight effectiveness can be – and typically is expected to be – highly context-sensitive. In general, empirical testing scales poorly because it requires effort to conduct empirical testing with actual human oversight personnel, and because it likely cannot be shifted from the token or deployment level to the type or provider licensing level due to the context-sensitivity of effective human oversight and the resulting loss of transferability of insights. Furthermore, deriving reliable conclusions often requires multiple studies (e.g., on the effects of different explainability approaches), suggesting that oversight requirements may need to be informed by high-quality meta-analyses that synthesize findings across studies for broader applicability.

Checklist-based approaches and empirical approaches are clearly not the only possible options for testing compliance with human oversight requirements, but they illustrate a fundamental tension in AI governance. Traditional checklist approaches offer efficiency and standardization but risk treating oversight as a technological feature to be verified rather than a sociotechnical capability to be validated. They may produce inconclusive results, such as when oversight mechanisms are formally in place but key performance indicators conflict, when standard checklist items cannot capture context-specific adequacy, or when there are gaps between documented policies and actual practice. Empirical approaches, by contrast, can validate the actual effectiveness of human oversight but are resource-intensive and context-sensitive and thus difficult to standardize.

This tension reflects the broader challenge of shifting from evaluating "good" technological products against standardized criteria to validating "good" sociotechnical systems where effectiveness emerges from dynamic interactions between humans, technology, and specific contexts. Hybrid approaches may offer a way forward: when checklist-based testing produces inconclusive results or reveals gaps between formal compliance and effective oversight, empirical validation could help bridge the measurement-reality divide. Clearly, such hybrid approaches raise their own questions about standardization as well as practical challenges such as resource allocation regarding compliance testing.

Furthermore, any form of checklist-based and empirical approaches faces additional challenges. First, it remains unclear when and how frequently human oversight processes need to be reevaluated. Should compliance assessments be conducted regularly or should they be triggered by evidence of non-compliance? Is it required to reevaluate compliance after each AI system update or after personnel changes?

Second, not only is the effectiveness of human oversight context-sensitive, the human oversight requirements themselves may also be context-dependent. For instance, oversight requirements may vary depending on the risk associated with the application context. Stricter oversight requirements may apply to AI used in the public sector compared to the private sector. Additionally, the required skills, expertise, and tasks of human oversight personnel can also vary significantly [7]. In real-time contexts, such as for the oversight of autonomous vehicles, sustained vigilance over long periods may be necessary, whereas in areas like hiring and credit scoring, human oversight personnel may require training in ethical and moral reasoning.

Third, certain regulatory requirements may be difficult to translate into testable standards for two key reasons. First, some requirements involve psychological factors that are challenging to operationalize and assess. This includes assessing whether human oversight personnel sufficiently understand AI limitations or testing for possible automation bias of human oversight personnel [5, 6]. Second, AI regulations typically include requirements where there is disagreement about guiding definitions and no clear ground truth available. For instance, in the case of discrimination and fairness, no commonly agreed standard exists for determining when a decision is discriminatory or unfair [4, 13]. This normative uncertainty, combined with the challenges of measuring psychological factors, helps explain why compliance testing particularly struggles with requirements related to transparency, explainability, and fairness [7].

These challenges point to the even more fundamental challenge that it remains unclear what constitutes sufficiently effective human oversight. Figure 1 illustrates this challenge. One fundamental expectation is that human oversight personnel add something beneficial to AI system operations. In other words, and in line with research and AI governance efforts using this term heavily [14], adding human oversight should increase the *trustworthiness* of AI operations. While trustworthy AI has many dimensions [8, 14, 15], one key goal – explicitly mentioned in the AI Act – is to increase safety. The effectiveness of human oversight depends on characteristics of oversight personnel (e.g., skills, training), the technology (e.g., AI transparency), and the operational context (e.g., roles, tasks, organizational factors) [9]. While it seems obvious that one key goal is that human oversight should make AI operations safer and more secure than autonomous operation of AI, determining when oversight is sufficiently safe and secure remains an open question.

Without concrete and testable standards for effective human oversight, normative uncertainty will persist for providers and deployers of AI regarding their legal compliance. This issue may be particularly pronounced for small businesses that lack the resources to establish an AI compliance department [7]. This uncertainty could lead to situations where the implementation of AI-based systems will be hampered in such businesses. Moreover, without concrete standards, virtually any implementation could be considered compliant [16]. This issue is especially problematic when audits rely on post-hoc rationalizations of human oversight implementations. In hindsight, any approach to human oversight could be justified as the "sufficiently good" or even "best possible" option.

3 Concluding Thoughts and Next Steps

The challenges of testing for compliance with human oversight requirements reflect broader difficulties in sociotechnical AI governance. As AI governance shifts from ensuring "good" technological products to "good" sociotechnical systems, defining standards will be complex, particularly when psychological concepts are involved. Beyond human oversight requirements, another key example relates to emotion recognition systems

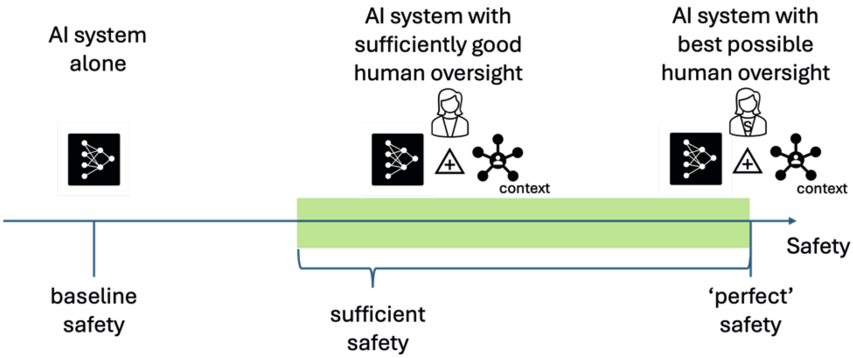


Fig. 1. A plausible criterion for effective human oversight is that human oversight of AI achieves a higher level of safety compared to the AI system operating alone (*baseline safety*). However, assessing safety has several layers: first, safety is inherently multi-dimensional, involving multiple, sometimes interdependent risks (e.g., an AI system may exhibit different forms of unfairness that cannot all be simultaneously eliminated). Second, even within a single safety dimension, it remains unclear what "perfect" safety would entail, making it particularly difficult to define or measure what counts as "sufficient" safety. This figure additionally highlights that safety (much like all dimensions of the effectiveness of human oversight; [9]) depends on characteristics of oversight personnel (e.g., skills, training), the technology (e.g., AI transparency), and the operational context (e.g., roles, tasks, organizational factors).

[17, 18]. According to the AI Act, AI systems that automatically infer emotions (e.g., sadness) in high-risk contexts are prohibited but inferring physical states (e.g., fatigue) is permitted. This raises questions such as: Is fatigue a purely physical state from lack of sleep or a symptom of depression? If linked to depression, would its detection be permitted? These questions seem relevant to governing respective AI products but they can only be adequately addressed by taking a sociotechnical perspective that addresses the intricate interplay between technical design (e.g., what model is used to infer emotions?), human factors (e.g., how do verbal, nonverbal, and paraverbal behavior relate to emotions?), and contextual considerations (e.g., what inferences about emotions are adequate at different workplaces?).

As outlined before, additional difficulties arise when AI governance seeks to mitigate risks for which no clear ground truth exists (e.g., risks of discrimination [7]) or when it remains uncertain whether risks have been effectively mitigated. For instance, was a fairness monitoring tool truly successful if it detects only one specific type of fairness violation in AI outputs [4, 19, 20]? Again, this emphasizes the need for a sociotechnical perspective that addresses technical design (e.g., for what kind of fairness metric to calibrate AI-based decisions), human factors (e.g., training in the detection of fairness issues), and contextual considerations (e.g., what kind of fairness definition is appropriate in a given context)?

The next steps in testing for compliance with AI regulation are currently being developed. Standardization and norming bodies are working to operationalize the requirements outlined in AI regulation. We anticipate that emerging standards and norms will fuel the debate on how to effectively test for compliance. The challenges outlined in this article will continue to require input from researchers, practitioners, and policymakers to ensure that AI governance effectively reduces the risks associated with AI systems while enhancing safety in their implementation without placing an undue burden on providers and deployers through resource-intensive, context-dependent compliance testing.

In the case of human oversight, key tasks for the near future are to (a) establish a middle ground or a feasible combination between checklists and empirical testing, (b) develop standards and norms that are informed by and adapt to the latest research in human-computer interaction, psychology, and related fields on effective human oversight [21], such as methods for preventing automation bias or effectively preparing and supporting humans to detect inaccurate and problematic outputs, and (c) evaluate the impact of human oversight requirements in AI practice. Finally, we want to highlight the crucial importance of expertise on the human factor in human-AI interaction for designing and testing for the effectiveness of human oversight. As AI governance evolves beyond technological improvement to optimizing sociotechnical systems for high-risk tasks, we believe research(ers) from psychology, human-computer interaction and related fields should play a key role in providing insights on how to optimize the technology, how to design the jobs and environments where humans and AI-based systems interact, and how to prepare and support human oversight personnel. This ensures that expertise and perspectives on the human factors in AI augments the expertise and perspectives that already play a key role in AI governance such as legal sciences, ethics, machine learning and other technical and engineering perspectives. This would then also substantiate commonly phrased claims (particularly in Europe) that AI governance aims for "human-centered AI" implementation that in practice often lacks to consult and integrate expertise on the "human" factors.

Acknowledgments. This work was partially funded by DFG grant 389792660 as part of TRR 248 CPEC – Center for Perspicuous Computing (Kevin Baum & Markus Langer), by the project TITAN – Technologische Intelligenz zur Transformation, Automatisierung und Nutzerorientierung des Justizsystems funded by the Daimler and Benz Foundation (grant no. 45-06/24) (Markus Langer), by the European Regional Development Fund (ERDF) and the Saarland within the scope of the project (To)CERTAIN – Towards a Center for European Research in Trusted AI, and received support from the German Federal Ministry of Education and Research (BMBF) as part of the project MAC-MERLin (Grant Agreement No. 01IW24007) (Kevin Baum).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Appendix A

Content of Article 14 of the European AI Act: Human Oversight

(1) High-risk AI systems shall be designed and developed in such a way, including with appropriate human-machine interface tools, that they can be effectively overseen by natural persons during the period in which they are in use.
(2) Human oversight shall aim to prevent or minimise the risks to health, safety or fundamental rights that may emerge when a high-risk AI system is used in accordance with its intended purpose or under conditions of reasonably foreseeable misuse, in particular where such risks persist despite the application of other requirements set out in this Section.
(3) The oversight measures shall be commensurate with the risks, level of autonomy and context of use of the high-risk AI system, and shall be ensured through either one or both of the following types of measures:
    (a) measures identified and built, when technically feasible, into the high-risk AI system by the provider before it is placed on the market or put into service;
    (b) measures identified by the provider before placing the high-risk AI system on the market or putting it into service and that are appropriate to be implemented by the deployer.
(4) For the purpose of implementing paragraphs 1, 2 and 3, the high-risk AI system shall be provided to the deployer in such a way that natural persons to whom human oversight is assigned are enabled, as appropriate and proportionate:
    (a) to properly understand the relevant capacities and limitations of the high-risk AI system and be able to duly monitor its operation, including in view of detecting and addressing anomalies, dysfunctions and unexpected performance;
    (b) to remain aware of the possible tendency of automatically relying or over-relying on the output produced by a high-risk AI system (automation bias), in particular for high-risk AI systems used to provide information or recommendations for decisions to be taken by natural persons;
    (c) to correctly interpret the high-risk AI system's output, taking into account, for example, the interpretation tools and methods available;
    (d) to decide, in any particular situation, not to use the high-risk AI system or to otherwise disregard, override or reverse the output of the high-risk AI system;
    (e) to intervene in the operation of the high-risk AI system or interrupt the system through a 'stop' button or a similar procedure that allows the system to come to a halt in a safe state.
(5) For high-risk AI systems referred to in point 1(a) of Annex III, the measures referred to in paragraph 3 of this Article shall be such as to ensure that, in addition, no action or decision is taken by the deployer on the basis of the identification resulting from the system unless that identification has been separately verified and confirmed by at least two natural persons with the necessary competence, training and authority.

The requirement for a separate verification by at least two natural persons shall not apply to high-risk AI systems used for the purposes of law enforcement, migration, border control or asylum, where Union or national law considers the application of this requirement to be disproportionate.

References

1. Lam, K., Lange, B., Blili-Hamelin, B., Davidovic, J., Brown, S., Hasan, A.: A framework for assurance audits of algorithmic systems. In: The 2024 ACM Conference on Fairness, Accountability, and Transparency, Rio de Janeiro Brazil: ACM, pp. 1078–1092 (2024). https://doi.org/10.1145/3630106.3658957
2. Mökander, J., Axente, M., Casolari, F., Floridi, L.: Conformity assessments and post-market monitoring: a guide to the role of auditing in the proposed European AI regulation. Mind. Mach. **32**(2), 241–268 (2022). https://doi.org/10.1007/s11023-021-09577-4
3. Green, B.: The flaws of policies requiring human oversight of government algorithms. Comput. Law Secur. Rev. **45**, 105681 (2022). https://doi.org/10.1016/j.clsr.2022.105681
4. Langer, M., Baum, K., Schlicker, N.: Effective human oversight of AI-Based systems: a signal detection perspective on the detection of inaccurate and unfair outputs. Minds and Machines (2024). https://doi.org/10.1007/s11023-024-09701-0
5. Ruschemeier, H., Hondrich, L.: Automation bias in public administration - an interdisciplinary perspective from law and psychology. SSRN Journal (2024). https://doi.org/10.2139/ssrn.4736646
6. Laux, J.: Institutionalised distrust and human oversight of artificial intelligence: towards a democratic design of AI governance under the European Union AI Act. AI Soc. **39**(6), 2853–2866 (2024). https://doi.org/10.1007/s00146-023-01777-z
7. Scantamburlo, T., et al.: Software systems compliance with the AI act: lessons learned from an international challenge. In: Proceedings of the 2nd International Workshop on Responsible AI Engineering, Lisbon Portugal: ACM, pp. 44–51 (2024). https://doi.org/10.1145/3643691.3648589
8. Schlicker, N., Uhde, A., Baum, K., Hirsch, M.C., Langer, M.: How do we assess the trustworthiness of AI: introducing the trustworthiness assessment model (TrAM). Comput. Hum. Behav. (2025). https://doi.org/10.31234/osf.io/qhwvx
9. Sterz, S., et al.: On the quest for effectiveness in human oversight: Interdisciplinary perspectives. In: The 2024 ACM Conference on Fairness, Accountability, and Transparency, Rio de Janeiro Brazil: ACM, pp. 2495–2507 (2024). https://doi.org/10.1145/3630106.3659051
10. Arrieta, A.B., et al.: Explainable artificial intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. Inf. Fusion **58**, 82–115 (2020). https://doi.org/10.1016/j.inffus.2019.12.012
11. Langer, M., et al.: What do we want from explainable artificial intelligence (XAI)? A stakeholder perspective on XAI and a conceptual model guiding interdisciplinary XAI research. Artif. Intell. **296**, 103473 (2021). https://doi.org/10.1016/j.artint.2021.103473
12. Lai, V., Chen, C., Smith-Renner, A., Liao, Q.V., Tan, C.: Towards a science of human-AI decision making: an overview of design space in empirical human-subject studies. In: 2023 ACM Conference on Fairness, Accountability, and Transparency, Chicago IL USA: ACM, pp. 1369–1385 (2023). https://doi.org/10.1145/3593013.3594087
13. Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., Galstyan, A.: A survey on bias and fairness in machine learning. ACM Comput. Surv. **54**(6), 115 (2021). https://doi.org/10.1145/3457607
14. Laux, J., Wachter, S., Mittelstadt, B.: Trustworthy artificial intelligence and the European Union AI act: on the conflation of trustworthiness and acceptability of risk. Regul. Gov. **18**(1), 3–32 (2024). https://doi.org/10.1111/rego.12512
15. High Level Expert Group on Artificial Intelligence. Ethics guidelines for trustworthy AI. Brussels, Belgium: European Commission (2019). https://ec.europa.eu/newsroom/dae/document.cfm?doc_id=60419

16. Cobbe, J., Lee, M.S.A., Singh, J.: Reviewable automated decision-making: a framework for accountable algorithmic systems. In: Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event Canada: ACM, pp. 598–609 (2021). https://doi.org/10.1145/3442188.3445921
17. Prégent, A.: Is there not an obvious loophole in the AI act's ban on emotion recognition technologies? AI Soc. s00146-025-02289–8 (2025). https://doi.org/10.1007/s00146-025-02289-8
18. Meinel, P., Lauber-Rönsberg, A.: 'How do you feel?' – Emotionserkennung nach der KI-VO. Datenschutz Datensich **49**(4), 236–240 (2025). https://doi.org/10.1007/s11623-025-2078-3
19. Biewer, S., et al.: Software doping analysis for human oversight. Formal Methods Syst. Design (2024). https://doi.org/10.48550/arXiv.2308.06186
20. Baum, K., et al.: Taming the AI monster: monitoring of individual fairness for effective human oversight. In: Model Checking Software, vol. 14624, Neele, T., Wijs, A. (eds.) in Lecture Notes in Computer Science, vol. 14624. , Cham: Springer Nature Switzerland, pp. 3–25 (2025). https://doi.org/10.1007/978-3-031-66149-5_1
21. Laux, J., Ruschemeier, H.: Automation bias in the AI Act: on the legal implications of attempting to de-bias human oversight of AI (2025). arXiv. https://doi.org/10.48550/ARXIV.2502.10036

Supporting a SOTIF Safety Argument by Activation Pattern Monitoring with Statistical Guarantees

Rüdiger Ehlers(✉), Loich Kamdoum Deameni, and Nikita Maslov

Clausthal University of Technology, Clausthal-Zellerfeld, Germany
ruediger.ehlers@tu-clausthal.de

Abstract. Modern autonomous-driving solutions rely on neural networks for visual perception. They typically lack precise specifications for when their behavior is considered to be correct, which complicates the use of traditional specification-driven verification approaches. To address this challenge, ISO standard 21448 ("Safety of the Intended Functionality", SOTIF) proposes activities focused on reducing – rather than eliminating – the risk of using machine-learned models and the resulting extent of harm.

One valuable activity in a SOTIF-based development process is runtime monitoring, as it provides a safeguard against scenarios that could not be anticipated during development. In the context of visual perception components based on learned neural networks, a runtime monitor can detect previously unknown driving scenarios during operation. For a SOTIF-based safety argument, however, the value it brings to the table needs to be quantified.

In this paper, we show how by combining activation pattern monitoring with ideas from conformal testing, a monitoring approach with statistical guarantees can be defined that supports a SOTIF safety argument. We apply an ellipsoid-based abstraction of the activation patterns that are local to the output of a YOLO real-time object-detection neural network. We demonstrate that by restricting the scope of the monitor to detect input that is clearly out-of-domain (OD) at runtime, a high accuracy of the monitor can be obtained, leading to strong safety guarantees that a SOTIF safety argument can build on.

1 Introduction

In current autonomous driving solutions, AI-based system components take safety-critical roles. Most notably, artificial neural networks are widely used for camera-based environment perception components of autonomous vehicles. Unlike for many traditional engineering activities, the development process of such networks does not start from a verifiable specification. This is rooted in the fact that the absence of a good model for how exactly the different types of objects to be perceived are represented in the image is the reason for using AI

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, pp. 170–185, 2026.
https://doi.org/10.1007/978-3-032-07132-3_12

in the first place. As a consequence, traditional specification-driven approaches for building safe systems are not applicable.

So how can, under these conditions, the sufficient safety of such AI-based components be proven and communicated to the general public, as needed for the societal acceptance of autonomous driving in the long run? The approach followed in the ISO standard 21448 ("*Safety of the Intended Functionality*" – SOTIF) to address this question is to focus on the engineering process of the component rather than only on the result of the process. This standard defines a set of activities that includes identifying potential hazards, their triggering conditions, and developing scenarios in which the hazard can lead to harm. By performing these activities, during the development process, previously unknown scenarios in which the system needs to operate correctly in become identified as "known" (w.r.t. the development process), and the percentage of scenarios in which the system behaves safely also increases. At the end of the process, all activities performed together need to support an argument showing that the system is both safe enough in known scenarios as well as in unknown scenarios. The latter case addresses the problem that it is not possible to test AI-based systems in all situations that can occur in the field since it is not even completely well-defined what exactly constitutes a scenario. Similarly, formal verification is not applicable because a verification process would require a precise model of the input to which the system is applied.

Proving the sufficient safety of a system in unknown scenarios appears to be contradictory at first, as for evaluating a system in a scenario, the scenario has to be defined. This contradiction is resolved by the respective chapter on the evaluation of unknown scenarios of the ISO standard. It clarifies that this type of evaluation is concerned with activities that augment classical testing on defined scenarios. This can for instance be the analysis of the effect of input noise on a learned model, the behavior of a system under consideration on random test cases, or the analysis of corner cases. Hence, the evaluation on unknown scenarios is concerned with analysis steps for which it is reasonable to expect that good performance of the system correlates with safe behavior in the field. As it is societally expected that autonomous driving systems implement the *state of the art* in such activities and at the end of these activities, they need to contribute to an overall safety argument, it is fair to expect a *reasonable* baseline of such activities commonly used in development processes according to ISO 21448 to develop over time.

In this context, *runtime monitoring* is a particularly attractive activity because it solves two problems at the same time, namely that (a) not all situations can be considered up-front and before the deployment of the system, and (b) runtime monitoring makes *what* is being monitored explicit. Given that the property being monitored is reasonable, continuous monitoring of an AI-enabled system component can increase the trust in the component. In a nutshell, a runtime monitor observes the behavior of a system and raises an alarm when the property whose satisfaction it monitors is violated. While due to the absence of a precise specification of the system, false positives and false negatives are impossible to avoid in their entirety, the output of a monitor can be used as an indicator for whether the driving situation is suitable for the autonomous driving function. This is especially the case when the output of a monitor is used to inform the system about whether the current situation is potentially not fully tested (e.g., by disabling overtaking on roads if it is questionable whether the current driving situation is fully understood by the perception system). Furthermore, the monitor output can be used to decide in which cases the input to the system should be recorded so that it can be used as new situation in the next iteration of the system's development process.

Given the conceptual simplicity of monitoring, it is not surprising that a plethora of monitoring approaches for a multitude of types of properties of interest have emerged. These include, most prominently, detecting out of distribution (OOD) input [17,24] to a machine-learned model. In this context, *out of distribution* refers to the input not being in the set of inputs from which the training data of the machine-learned model was taken, ideally via random selection. Then, there are monitoring approaches that concern whether the system under inspection adheres to expected aspects of their input/output behavior [1]. As a final example, one can monitor for whether the *activation patterns* that an artificial neural network computes for some layers of its architecture at runtime are included in some abstractions of the set of patterns on which the system has shown correct behavior [5]. Such an activation pattern is just the output computed by one (typically late) layer of a neural network.

The last of these examples is somewhat surprising at first, because it concerns the *internal* state of an AI-based system. The idea behind this type of monitoring is that whenever an activation pattern is atypical (i.e., very different to what has been observed on the training data), then the system behavior should not be trusted as it was not *trained* on these activations. In this way, activation pattern monitoring can also be seen as a technique for OOD detection. While the relationship between activation patterns and input/output behavior of the AI-based system is rather indirect, this approach to monitoring has an interesting property: it is close to traditional monitoring of engineered systems. If a system is engineered with certain *invariants* in mind (such as the temperature in a machine being less than 100°C), then operating it outside of these invariants is risky. Activation pattern monitoring mimics sanity checks of sensor data in regular embedded systems: if some measured values are implausible, the system needs to fall back to safer (but probably suboptimal) behavior. With these connections, activation pattern monitoring can play a part in *explaining* the safety of a system. To be truly useful to support a safety argument, the value that activation pattern monitoring brings to the table needs to be properly quantified, however, which classical activation pattern monitoring approaches from the field of formal methods do not.

In this paper, we revisit activation pattern monitoring and combine it with a clearly defined purpose in the context of supporting a safety argument. We show how by integrating ideas from the field of *conformal prediction* to obtain statistical guarantees on the *perception system operating within the input space it was designed for*, we can support a SOTIF-based development process. In particular, we check, with statistical confidence, whether the system's state has been pushed outside of what it was designed to work on due to the system being operated outside of its *operational design domain* (ODD), i.e., the set of situations and environments it was designed for. The statistical guarantees are based on the assumption that data used in the development process is drawn in a representative way from the scenarios of interest in the real world, so that such a monitoring process can provide a quantitative component of a safety argument. In particular, we provide the following contributions:

- We provide a summary of the differences and similarities of activation pattern monitoring to classical OOD and ODD detection.
- We discuss which formulations of the activation pattern monitoring problem are reasonable from a conceptual point of view.
- We provide an approach to define *local* neighborhoods for activation pattern monitoring in a *YOLO*-like object detection network.
- We show how to adapt some ideas from conformance prediction to add statistical guarantees to activation pattern monitoring.
- Finally, we show some results on an activation pattern monitoring example setting that demonstrate that this approach can detect *out of domain* activation patterns.

The resulting monitor can be used in a safety argument by (a) making the autonomous driving system work with degraded performance if the monitor triggers more than intermittently, and (b) making a probabilistic argument on the completeness of engineering process by connecting how often the monitor triggers in field tests with the statistical guarantees that it provides. While the statistical argument is based on the assumption of representativeness of the data used for calibrating the guarantee, the rigor used in the data collection process of a SOTIF-based development process can be used to support this claim, contributing to a proof that the residual risk of using the system is as low as reasonably possible.

2 Related Work

Activation pattern monitoring of artificial neural networks [6,12,23], also called abstraction-based monitoring, has its roots in formal methods. Such approaches supervise the output of a layer of an artificial neural network and check if it lies in an *abstraction* of the set of patterns found to be typical of normal operation.

Activation pattern monitoring has multiple purposes. On the one hand, it has been noted that in practice, an activation pattern being outside of the computed abstraction correlates with the input of the overall network being out-of-distribution (OOD) [23]. On the other hand, Lukina et al. [14] use abstraction-based monitoring to decide when enough novel input has been fed to the neural network so that a manual data labeling and training process should be triggered. They emphasize that in this way, monitoring helps with making the operation of the network more transparent to the user. Furthermore, they show in their experiments that the monitor results can be used to detect input that requires manual labeling, hence optimizing the retraining process of the system when the budget of allowed interaction with a human for manual labeling is small.

Boursinos and Koutsoukos [2] combine the idea of monitoring activation patterns with *conformance prediction*, as we do in this paper. Conformance prediction is the process of postprocessing the output of a learned classification model to a *set* of classes such that a statistical guarantee can be given that the class that is correct for the given input is contained in the set. Just as in this paper, their approach detects mispredictions with statistical confidence. They deviate in their aim from our work, however: their purpose is to detect mispredictions, whereas we focus on detecting whether the AI model is being used outside of its operational design domain such that the internal state (an activation of a late layer) is outside of what the model has been trained for. This aim is more modest, resulting in the possibility to obtain a high accuracy. By combing a high reliability with a modest aim, such a monitor can fulfill a well-defined purpose in an overall system safety argument. Also, our approach is, unlike the one by Boursinos and Koutsoukos, not based on considering different activation pattern set abstractions for the individual classes, but detects regions in an input image for which all classifier outputs together should not be trusted.

Activation pattern monitoring is conceptually related to *out of distribution* (OOD) input detection as well as monitoring the *operational design domains* (ODD) of an AI-based systems. Ultimately, the goal of ODD detection is to identify input that is somewhat distant to the training dataset, and there is no precise definition of what exactly this means, given that the probability distribution of the input to expect is not accessible for model training. A typical concretization in this context is to treat this problem as a one-class classification problem (in the absence of atypical input data), and for instance define approaches to classify the probability distributions that form the output of a classification artificial neural network into expected and unexpected ones [21]. Other approaches operate on the input directly [11]. Guérin et al. [9] argue that the problem of detecting out-of-distribution input is ill-focused by the fact that extrapolating to untrained cases is the whole point of learning a model in the first place. They propose moving to a two-class classification problem for detecting "Out-of-Model-Scope (OMS)" cases, which are defined by the learned model mispredicting in them.

In contrast, what we aim for here is a simple sanity checking argument of the system's internal state that is close to monitoring the operating conditions of technical systems, while addressing the fact that the state of an AI-based perception system is an activation pattern. Although we adapt the proposal to treat the problem at hand as a two-class classification problem (in Sect. 3.2), we do not explicitly consider whether the network misclassifies. Instead, we are interested in detecting inputs on which the AI model is not applicable and no correct classification even exists. The simplicity of the approach in combination with a statistical guarantee makes it suitable to support a formal safety argument while allowing to built trust in a sufficiently low error rate of the system.

While it would intuitively make sense to treat the problem solved in this paper as an instance of operational design domain (ODD) monitoring (see, e.g., [4]), the term is normally used for reasoning over the expected input at a higher level, such as how many vehicles can be visible at the same time to a perception system. Such aspects are difficult to translate to the concrete input to such a system apart from sampling potential input by means of a simulator [7].

Few works consider the problem of detecting OOD or ODD data in the scope of a system constructed using a SOTIF-based development process. Hacker and Seewig [10] recently considered a monitor ensemble for identifying several types of safety-related insufficiencies in perception systems for autonomous vehicles. For OOD detection, they use a heuristic that involves comparing multiple learned models, which complicates making a statement about the scope of the monitor. Also, their monitor ensemble aims at identifying multiple types of insufficiencies rather than supporting an individual aspect of the overall safety argument.

3 How to Detect Cases in Which a System Was Not Designed to Operate In?

When addressing the challenge of detecting whether an AI perception system is operating within the conditions it was designed for, the first aspect to clarify is what this actually means. For the scope of this paper, we consider the problem of detecting objects in the scope of an AI model operating according to the YOLO (*you only look once*) principle [15,22]. YOLO-based models operate on a whole input image, but the image is partitioned into a grid of cells, and the same convolutional neural network is applied to each cell. Figure 1 shows an illustration of the partitioning of an input image. For each cell, the model predicts a) a number of values that are used to determine the x and y positions as well as the width and height of the bounding box of a potential object in the cell, b) one class probability for every object class used in the dataset (which indicates the likelihood of the cell containing an object of this class), and c) an overall probability of the cell containing an object. Furthermore, YOLO models often use several grid granularities of the input at the same time in order to detect objects of different sizes. Finally, all probabilities and potential bounding box positions are post-processed by a so-called *header* to produce the actual bounding boxes of the detected objects.

Feeding input to the model that is from a different domain (*off domain*, OD) than the data on which the model was trained (*in domain*, ID) can make the input to the header meaningless, leading to the model being *blind* on some of the parts of their input image even if the respective part contains objects from the classes considered during training. For a responsible use of YOLO models in the field, we hence have to balance the expectation that such models extrapolate from learned data to a new slightly different input in the field with the purpose of detecting when the model operates in a domain it was not designed for. For instance, when some cells of the input image show a poster board at the side of the road or a truck tarpaulin (possibly with a picture printed on it), or a part

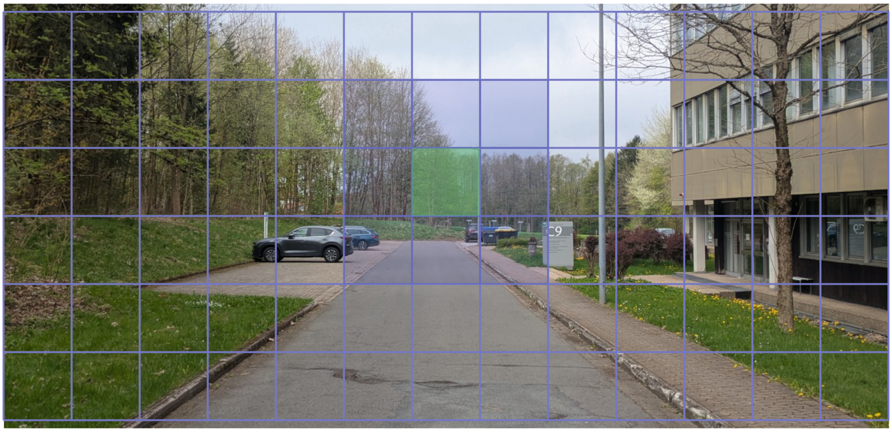


Fig. 1. Example input image and its split into cells.

of the lens is covered with dirt or a sticker, then the model under consideration is blind on these input cells.

In such cases, it is reasonable to expect artificial neural networks to exhibit atypical activation patterns. In fact, neural networks learn specific activation patterns that allow them to correctly classify inputs on which it was trained. However, the standard training process of a model does not explicitly prevent the output of individual neurons from deviating from their usual range for inputs that are off-domain (OD). Cheng et al. [6] experimentally validated this effect in their work on activation pattern monitoring. Building on this observation, our goal is to detect when, for *parts* of an input image, the system operates outside of its intended boundaries.

For this purpose, we consider *local activation patterns* while focusing on a single grid scale. The general idea is visualized in Fig. 1 – the neural network output for a 3×3 blocks large input group is used in order to estimate whether the cell in the center of the group contains image data on which the perception system operates outside of its domain. We aggregate the object class probabilities for each cell in the group to a vector (activation pattern) on which the monitoring approach described in the following operates.

Performing the monitoring process in this way has the property that it can support a safety argument: the activation patterns are simple to measure "sensor values" about the internal state of the system and treating the input in blocks allows to pinpoint for parts of an input whether the system's output can be trusted on it – in this way, the detection of blind spots can be reasoned over in the system's safety argument based on the monitor output.

3.1 A First Experiment Using One-Class Classification

For a first experiment, we considered a YOLOv3 model [16] on the KITTI dataset [8] with traffic situations trained on detecting objects from 96 classes, leading to activation vectors with 964 elements each. We overall extracted 82962 activation patterns from 3771 images. We also use 27676 patterns from 1258 additional images from the same dataset for testing whether the monitor detects these as typical for the domain.

Henzinger et al. [12] proposed to use "boxes" (axis-parallel hyperrectangles) for abstracting activation patterns from the training data to a representative set of activation data. The core idea of their approach is that hyperrectangles encapsulate activation patterns of inputs on which the system was trained, and patterns not in the hyperrectangles are seen as suspicious.

They experimentally compared their box-based abstraction to two other methods – specially balls and octagons – based on their precision and computational complexity. Their experiments demonstrated that using boxes balances good precision in novelty detection with efficient computation. They concluded that using a box-based abstraction allows for effective runtime monitoring with minimal overhead. Although their boxes are simple to compute, they have the odd property that activations that are relatively extreme in many dimensions, but never the most extreme one in any dimension, might not be detected.

To address this limitation of boxes and to provide tighter boundaries than balls, with minimal computational overhead, we propose to use an *ellipsoid* representation. An ellipsoid can be encoded with a single positive definite matrix A and a vector b such that an activation pattern x is in the ellipsoid if and only if $(x - b) \cdot A \cdot (x - b)^T \leq 1$. The set of ellipsoids is closed under rotation, skewing, scaling, and translation, which is desirable from a conceptual point of view. In particular, as any rotation, translation, and skewing performed by a neural network layer can be undone by a subsequent linear layer, and there are no steps in the learning process that prevent the learning process from making unnecessary such skewings, translations, and rotations, it makes sense to use an activation pattern representation that is closed under such linear transformations.

We first considered the activation pattern monitoring problem as a one-class classification problem by computing an ellipsoid around all 82962 activation patterns in the training dataset. Using an implementation of Khachiyan's algorithm [20], we can compute an ellipsoid enclosing an approximately minimal volume. Doing so adds an quality criterion to the computed ellipsoid, which is needed to avoid trivial solutions (an ellipsoid containing all possible activation patterns for arbitrary input images). The amount of data considered in this experiment is however already too large for the algorithm to be applied (observed by it running out of memory when applying the algorithm on a computer with 1 TB of RAM). As a consequence, we experimented with a classical machine learning-based approach. Rather than learning the matrix A directly, we apply classical back-propagation based learning to learn an ellipsoid center, a size (in each dimension), and a fixed-length list of mirroring planes. Input data points are mirrored by the planes in order, and the resulting point is checked for containment in the (then) axis-parallel ellipsoid. With the mirroring planes, skewing, mirroring, and rotation can be learned while the number of planes provides a way to select the complexity of the learnable skewing and rotation of the ellipsoid.

Table 1. Performance results for the ellipsoid-based monitor from Sect. 3.1

# Planes	Patterns contained in ellipsoid		
	Training Data	Testing Data (ID)	Testing Data (OD)
0	99.2 %	99.2 %	75.8 %
10	99.2 %	99.2 %	86.2 %
80	99.2 %	99.3 %	95.3 %

In the learning process, we use the product of the ellipsoid sizes (in each dimension) as an additional optimization criterion in the *loss function* that guides the learning process. Doing so results in the volume enclosed by the ellipsoid to be minimized (heuristically) during learning. We experimented with several different numbers of mirroring planes, for which we give results in Table 1.

We observe that the learning process succeeds with more than 99% of the in-domain images being contained in the ellipsoid, both in the training data set as well as the testing data set from the application domain. To estimate the usefulness of the monitor in the field, we also considered 44143 activation patterns obtained from 521 images from the COCO dataset [13], which does not concern traffic situations (with some exceptions), as example input that is out-of-domain. Table 1 shows that most of the activation patterns from such inputs however fall into the ellipsoid. In particular, the more freedom we give to the learning process (by increasing the number of hyperplanes to mirror the activation patterns by), the more activation patterns from the out-of-domain dataset are contained in the ellipsoid. Figure 2 shows a histogram of the weighted distances (results of computing $(x - b)A(x - b)^T$ for the ellipsoid center b and the ellipsoid matrix A) of the activation patterns to the ellipsoid center for the ellipsoid learned without mirroring planes. It can be see that the distances of the ID and OD cases overlap heavily despite the distributions of these cases differing notably. For a quantitative safety argument in a SOTIF-based development process, these results appear to be insufficient, however. This observation leads to the question whether this weak performance is unavoidable for this type of monitor or whether a different type of learning process can improve it.

3.2 Treating the Out-of-Domain Detection Problem as a Two-Class Classification Instance

Guérin et al. [9] argue that treating OOD detection as a one-class classification problem is unreasonable as a primary goal of learning a model is to have it extrapolate to unseen data. Instead, they propose to view the problem as a two-class classification task where classes are defined by whether the learned model under concern behaves correctly or incorrectly on an input.

Building on this idea, we can reformulate the OD detection problem on activation patterns to learn a two-class classifier that distinguishes between patterns

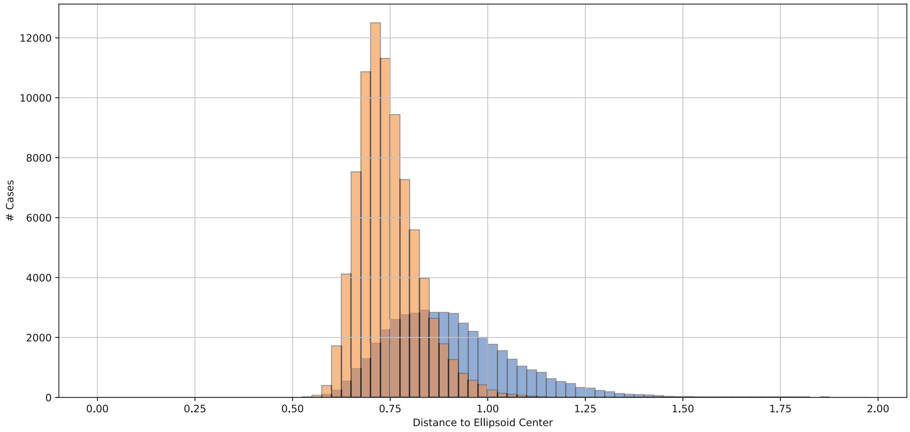


Fig. 2. Histogram of the weighted ellipsoid center distances for ID (orange) and OD (blue, more to the right) activations for the approach from Sect. 3.1 (Color figure online). The distances are normalized to 1 so that exactly those of at most 1 correspond to patterns in the ellipsoid.

from in-domain and off-domain input. For the following experiment, we adopt this approach and continue employing the ellipsoid representation.

We use an additional 1560 out-of-domain images from the COCO dataset for training the two-class classifier, resulting in 131241 activation patterns. We again learn an ellipsoid with a backpropagation-based learning process, but this time without size minimization for the ellipsoid.

Interestingly, the resulting classifier achieves a high accuracy on both the training data as well as the testing data, as shown in Table 2. As before, we experimented with a varying number of mirroring planes. The results show that both ID activation patterns and OD activation patters are detected in a reliable way. Upon closer examination of the results, it turns out that the problem to be solved is actually quite simple. Even without allowing the ellipsoid to be rotated or skewed, we obtain a model with 99 % of in-domain test cases and 99.13 % of out-of-domain test cases classified correctly. Figure 3 shows the histogram for the case without mirroring planes. It can be seen that the ellipsoid separates the ID and OD patterns relatively well despite the OD patterns still being relatively close to the ellipsoid center.

While the case without mirroring planes is relative plain, a resulting monitoring approach model still solves one problem of the box-based abstractions [14], namely that cases that are close to the boundaries in many dimensions at the same time are not detected as unusual.

Overall, this experiment shows that conceptually, two-class classification appears to be the more fitting approach for out-of-domain detection of activation patterns, at least for the purpose of detecting if the model output for parts of an input image are clearly out-of-domain. It should be noted that the images in the COCO dataset are very diverse, so that the comparison between

Table 2. Performance results for the ellipsoid-based monitor from Sect. 3.2

# Planes	Patterns contained in ellipsoid		
	Training Data	Testing Data (ID)	Testing Data (OD)
0	99.0 %	99.0 %	0.87 %
10	99.2 %	99.18 %	0.8 %

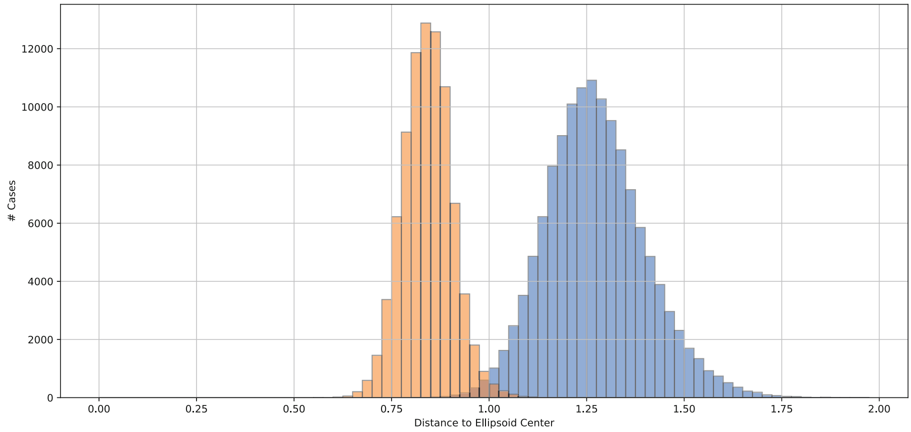


Fig. 3. Histogram of the weighted ellipsoid center distances for ID (orange, to the left) and OD (blue, to the right) activations for the approach from Sect. 3.1 (Color figure online). The distances are normalized to 1 so that exactly those of at most 1 correspond to patterns in the ellipsoid.

the one-class and two-class classification monitoring approaches is not skewed by similarities between images in the COCO dataset.

For a SOTIF safety argument, an ellipsoid-based monitor with a good accuracy is already useful. For instance, it can be used to detect novel input in the field for "phoning it home" to the perception system's manufacturer, also helping with human oversight of a vehicle manufacturer's fleet. Its high accuracy can serve as argument that the monitor-based preselection of new input to be phoned home balances the needs to keep the data transmitted by all systems of the manufacturer together reasonable while recording new situations that are relevant for the continuous engineering process of the system.

3.3 Adding Statistical Guarantees

The high accuracy on the relatively simple out-of-domain detection approach from above is useful for supporting a safety argument of a bigger system: while the scope of what is detected is very limited, the high accuracy can provide trust in the monitor detecting out-of-domain cases reliably. Furthermore, the documentation performed in the context of an ISO 21448 development process assigns the measures taken in the development process to individual types of hazards and risks – so the narrow scope of the monitor is actually rather an advantage than a drawback (which is somewhat at odds with the idea to evaluate approaches on difficult benchmarks sets, as common in works on OOD detection).

The overall safety argument for the perception system however needs to be quantitative, showing a high probability of working correctly across different situations. To support such an argument, the monitor should also provide statistical guarantees, which can be done by combining monitoring with ideas from *conformal prediction* [18]. Boursinos and Koutsoukos [3] already considered such a combination in the past, and we do so here in a way that makes use of the fact that ellipsoids induce a weighted distance metric from some mean activations. In this way, we can *calibrate* the monitor based on this metric to obtain guarantees.

More formally, we determine a distance threshold d_{limit} that allows us to identify in-domain activation patterns based on the calibration of a set of known out-of-domain activation patterns C, assumed to be independent and identically distributed (i.i.d.) drawn from the real distribution of the out-of-domain cases to be detected at runtime. More concretely, for each $x \in C$, we compute the distance $d(x) = (x - b) \cdot A \cdot (x - b)^T$ and aggregate them in a multiset M. Then, given a small probability of allowed failure α, we choose the greatest distance value d_{limit} such that at most a fraction of $\frac{(|C|+1)(1-\alpha)}{|C|}$ of the elements in M are below d_{limit}. Thus, if the monitor does not flag a (local) activation pattern as potentially problematic, then the probability that this verdict is correct is at least $1 - \alpha$ in the case of an out-of-domain activation pattern. In-distribution cases can however sometimes be detected as problematic even if they are not. At runtime, the resulting monitor considers all new activation patterns x' for which $d(x') \geq d_{limit}$ as *potentially* being out-of-domain. The correctness of this monitoring approach is based on the following theorem:

Theorem 1. *For any new activation pattern x' that is i.i.d. drawn from the distribution $Dist_C$ of out-of-domain input from which C was sampled, we have*

$$\mathcal{P}(d(x') \geq d_{limit} \mid x' \sim Dist_C) \geq 1 - \alpha$$

Proof. We follow the exposition by Shafer et Vovk [18] for a corresponding proof in the context of conformal prediction.

If we order the $|C| + 1$ distances computed for x' and the elements in C, the i.i.d. assumption implies that $d(x')$ is equally likely to appear anywhere in the ordered list. For any position $k \in \{1, \ldots, |C| + 1\}$, the probability for $d(x')$ to be among the k smallest elements in $M \cup \{d(x')\}$ is $\frac{k}{|C|+1}$. Therefore,

$$\mathcal{P}(d(x') \geq d_{limit}) = \frac{\lceil |C| \cdot \frac{(|C|+1)(1-\alpha)}{|C|} \rceil}{|C| + 1} = \frac{\lceil (|C| + 1)(1 - \alpha) \rceil}{(|C| + 1)} \geq 1 - \alpha. \qquad \square$$

We could, in a similar way, compute a value d'_{limit} such that, if a new input x is in domain, the probability of the weighted distance of its activation pattern from the center of the ellipsoid being at most d'_{limit} is at least $1 - \alpha$. In this way, we could detect out-of-domain cases with statistical guarantees. This approach is valid under the assumption that the in-domain patterns at runtime are i.i.d. drawn from the same distribution as the in-domain samples in the calibration dataset.

Let us evaluate monitoring with statistical guarantees on an example, where for simplicity we use the learned axis-aligned ellipsoid (without mirroring) from Sect. 3.2. We split the out-of-domain testing set into half calibration set and half post-calibration testing set for the monitor with statistical guarantees and choose $\alpha = 0.01$ for this example study. The calibration value is 1.0046 times the size of the ellipsoid for the monitoring process without statistical guarantees. From the OD activation patterns in the post-calibration testing set, 99.06 % were detected as potentially problematic. On the flip side, from the in-domain testing data, 0.87 % were detected as potentially out-of-domain.

Overall, it can be seen that the approach provides good performance on this rather small case study while giving statistical guarantees. In this way, the monitor can be used as a part of the overall safety argument, namely by addressing the need for a sufficiently high probability of an autonomous driving function detecting that it is used out of domain. The overall safety argument could then also reason over other monitors and/or the properties of error-resilient maneuver planning approaches using the perception systems' output as input.

4 Conclusion

This paper dealt with the question of how an activation pattern monitor can support making a safety argument for an AI-based environment perception system of an autonomous vehicle. When using a structured approach for doing so, such as the one in ISO 21448, every measure needs to have a clear scope and concrete quantifiable way in which it contributes to the overall safety argument. We showed that by setting the scope of such a monitor to detect activations resulting from far-from-normal local input, very high accuracy rates are possible on a moderately complex benchmark, and that also very strong statistical guarantees are possible. Key to this result was defining the scope of the monitor to be relatively narrow. While the next step is to evaluate the approach in an actual engineering context with more data, and it is conceivable that the activation pattern domain may need to be extended (e.g., to sets of ellipsoids) to maintain the good monitoring performance, the main focus of this paper was showing what role a simple activation pattern monitoring approach can take in a safety argument. Furthermore, the monitor output can be used to identify new input of interest so that it can be recorded for re-engineering the system. In this way, an activation pattern monitor can support human oversight of AI-based systems [19] of a manufacturer's fleet of vehicles, namely by pre-selecting cases on which the environment perception systems behavior should be checked post hoc. Finally, when the monitor results are fed to the driving maneuver decision making process, the monitor output can be used to avoid more advanced maneuvers (such as overtaking) when part of the input is likely to contain unrecognized objects.

At the same time, we also indirectly addressed the question of building societal trust in a system's correct operation beyond a formalized safety argument, namely by focusing on a monitoring process that is simple enough so it that it amounts to checking if some operating values stay within tested boundaries. This is a common self-supervision approach in mechanical and electrical engineering, and there is ample of experience in using it in these domains. While the monitoring approach described in this paper is, from a scientific point of view, unsophisticated, this property is also what enables wider use: engineers of safety-critical systems can only employ techniques that can be understood by a wider (engineering) audience in order to build a safety argument on them. We hope that by showing that with a narrow scope, activation pattern monitoring does not need to be complex, we provide an impulse on how this tension can be addressed in the field.

Acknowledgements. This work was partially funded by the Federal Ministry for Economic Affairs and Energy of Germany as part of the research project SafeWahr under grant number 19A21026E. The authors want to thank Ayrat Khalimov for his support with preparing this manuscript.

References

1. Aslam, I., Buragohain, A., Bamal, D., Aniculaesei, A., Zhang, M., Rausch, A.: A method for the runtime validation of AI-based environment perception in automated driving system. arXiv/CoRR **abs/2412.16762** (2024)
2. Boursinos, D., Koutsoukos, X.D.: Assurance monitoring of cyber-physical systems with machine learning components. arXiv/CoRR **abs/2001.05014** (2020)
3. Boursinos, D., Koutsoukos, X.D.: Assurance monitoring of learning-enabled cyber-physical systems using inductive conformal prediction based on distance learning. Artif. Intell. Eng. Des. Anal. Manuf. **35**(2), 251–264 (2021)
4. Charmet, T., Cherfaoui, V., Ibanez-Guzman, J., Armand, A.: Overview of the operational design domain monitoring for safe intelligent vehicle navigation. In: 26th International Conference on Intelligent Transportation Systems (ITSC), pp. 5363–5370. IEEE (2023)
5. Cheng, C., Luttenberger, M., Yan, R.: Runtime monitoring DNN-based perception - (via the lens of formal methods). In: 23rd International Conference on Runtime Verification (RV). Lecture Notes in Computer Science, vol. 14245, pp. 428–446. Springer (2023). https://doi.org/10.1007/978-3-031-44267-4_24
6. Cheng, C.H., Nührenberg, G., Yasuoka, H.: Runtime monitoring neuron activation patterns. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 300–303. IEEE (2019)
7. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: an open urban driving simulator. In: Proceedings of the 1st Annual Conference on Robot Learning, pp. 1–16 (2017)
8. Geiger, A., Lenz, P., Urtasun, R.: Are we ready for autonomous driving? The KITTI vision benchmark suite. In: Conference on Computer Vision and Pattern Recognition (CVPR) (2012)

9. Guérin, J., Delmas, K., Ferreira, R.S., Guiochet, J.: Out-of-distribution detection is not all you need. In: Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI, pp. 14829–14837 (2023). https://doi.org/10.1609/AAAI.V37I12.26732
10. Hacker, L., Seewig, J.: Insufficiency-driven DNN error detection in the context of SOTIF on traffic sign recognition use case. IEEE Open J. Intell. Transp. Syst. **4**, 58–70 (2023)
11. Hashemi, V., Kretínský, J., Rieder, S., Schmidt, J.: Runtime monitoring for out-of-distribution detection in object detection neural networks. In: 25th International Symposium on Formal Methods (FM). Lecture Notes in Computer Science, vol. 14000, pp. 622–634. Springer (2023)
12. Henzinger, T.A., Lukina, A., Schilling, C.: Outside the box: Abstraction-based monitoring of neural networks. In: ECAI 2020, pp. 2433–2440. IOS Press (2020)
13. Lin, T.Y., et al.: Microsoft COCO: common objects in context. In: Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds.) ECCV 2014. LNCS, vol. 8693, pp. 740–755. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10602-1_48
14. Lukina, A., Schilling, C., Henzinger, T.A.: Into the unknown: active monitoring of neural networks. In: 21st International Conference on Runtime Verification (RV), pp. 42–61 (2021)
15. Redmon, J., Divvala, S.K., Girshick, R.B., Farhadi, A.: You only look once: unified, real-time object detection. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 779–788 (2016)
16. Redmon, J., Farhadi, A.: YOLOv3: an incremental improvement. arXiv/CoRR **abs/1804.02767** (2018)
17. Salehi, M., Mirzaei, H., Hendrycks, D., Li, Y., Rohban, M.H., Sabokrou, M.: A unified survey on anomaly, novelty, open-set, and out of-distribution detection: solutions and future challenges. Trans. Mach. Learn. Res. **2022** (2022)
18. Shafer, G., Vovk, V.: A tutorial on conformal prediction. J. Mach. Learn. Res. **9**, 371–421 (2008). https://doi.org/10.5555/1390681.1390693
19. Sterz, S., et al.: On the quest for effectiveness in human oversight: interdisciplinary perspectives. In: The 2024 ACM Conference on Fairness, Accountability, and Transparency (FAccT), pp. 2495–2507 (2024)
20. Todd, M.J., Yıldırım, E.A.: On Khachiyan's algorithm for the computation of minimum-volume enclosing ellipsoids. Discret. Appl. Math. **155**(13), 1731–1744 (2007)
21. Wang, H., Li, Z., Feng, L., Zhang, W.: Vim: Out-of-distribution with virtual-logit matching. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4921–4930 (2022)
22. Wong, A., Famuori, M., Shafiee, M.J., Li, F., Chwyl, B., Chung, J.: YOLO NANO: a highly compact you only look once convolutional neural network for object detection. In: 2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS), pp. 22–25. IEEE (2019)
23. Wu, C., He, W., Cheng, C., Huang, X., Bensalem, S.: BAM: box abstraction monitors for real-time OoD detection in object detection. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 2632–2638 (2024)
24. Yang, J., Zhou, K., Li, Y., Liu, Z.: Generalized out-of-distribution detection: a survey. Int. J. Comput. Vision **132**(12), 5635 5662 (2024)

Author Index

B. Steffen (Ed.): AISoLA 2025, LNCS 16220, p. 187, 2026.
https://doi.org/10.1007/978-3-032-07132-3