Early Fault-Detection in the Development of Exceedingly Complex Reactive Systems

Assaf Marron[a] and David Harel[b]

Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, 76100, Israel

{assaf.marron, david.harel}@weizmann.ac.il

Keywords: System Engineering, Software Engineering, LLM, Models, Specifications, Interaction, Simulation, Verification, Autonomous Systems, Systems of Systems, Super-Reactive Systems.

Abstract: Finding hidden faults in reactive systems early in planning and development is critical for human safety, the environment, society and the economy. However, the ever growing complexity of reactive systems and their interactions, combined with the absence of adequate technical details in early development stages, pose a great obstacle. The problem is exacerbated by the constant evolution of systems, and by their extensive and growing interwoven-ness with other systems and the physical world. Appropriately, such systems may be termed *super-reactive*. We propose an architecture for models and tools that help overcome such barriers and enable simulation, systematic analysis, and fault detection and handling, early in the development of super-reactive systems. The main innovations are: (i) the allowing of natural language (NL) specifications in elements of otherwise standard models and specification formalisms, while deferring the interpretation of such NL elements to simulation and validation time; and (ii) a focus on early formalization of tacit interdependencies among seemingly orthogonal requirements. The approach is facilitated by combining newly specialized tools with standard development and verification facilities, and with the inference and abstraction capabilities of large language models (LLMs) and associated AI techniques. An important ingredient in the approach is the domain knowledge embedded in LLMs. Special methodological measures are proposed to mitigate well known limitations of LLMs.

1 INTRODUCTION

Since the 1985 identification of the category of *reactive systems*(Harel and Pnueli, 1984), a plethora of methods, languages and tools have been introduced to support the development of such systems. Today, complex reactive systems penetrate almost every aspect of life, including communications, commerce, finance, healthcare, aviation, land transportation, manufacturing, and more. The complexity of new systems is compounded by the fact that they are interwoven with other systems and with the physical world, and are constantly changing and evolving.

In this paper, we term this kind of system as *super-reactive* (SR). While system and software engineering (SySE) is benefitting from new developments in generative AI and large language models (LLMs), the challenge of building safe and reliable SR systems remains open. Despite applying the best tools and methodologies, any given system is likely to conceal undesired and very often unsafe behaviors and impending failures, with the risk of adverse effects on human life, the environment, society and the economy. Thus, while early discovery and handling of such faults is required, it remains a tantalizing challenge, growing alarmingly in severity as SR systems grow in complexity.

In this position paper, we propose a way of tackling this issue based on the following principles: (1) Allow model elements that are expressed in natural language (NL), benefitting from the expressive power of NL, its sensitivity to delicate context variations and its ability to navigate multiple levels of abstraction, and carrying out just-in-time (JIT), deferred, interpretation of such NL elements. (2) Discover and document otherwise-tacit interdependencies among separately specified, seemingly orthogonal requirements.

A key enabler for our approach is the availability of large language models and other AI tools, the power and breadth of which is also ever-growing. In section 4 we propose steps that can help circumvent known weaknesses in AI and LLM techniques.

[a] https://orcid.org/0000-0001-5904-5105
[b] https://orcid.org/0000-0001-7240-3931

Marron, A. and Harel, D.
Early Fault-Detection in the Development of Exceedingly Complex Reactive Systems.
DOI: 10.5220/0013369200003896

In *Proceedings of the 13th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2025)*, pages 321-329
ISBN: 978-989-758-729-0; ISSN: 2184-4348

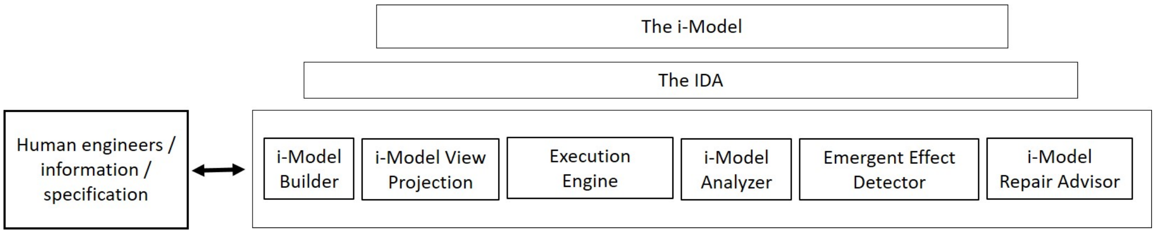

Figure 1: Solution architecture blueprint. See explanation in text.

2 THE PROBLEM

As the complexity and pervasiveness of reactive systems and systems of systems keeps growing, so do the risks associated with hidden faults: potential failure points, malfunctions, undesired behaviors and absence of desired ones. Recent well-known cases involving actual system code, maintenance procedures, interfaces with other systems and with humans, etc., include: the Crowdstrike server failures in July 2024 (George, 2024), the 2023 accident in which a robotic taxi hit a pedestrian in San Francisco(Koopman, 2024), the failure of the USA FAA notification system in 2023 (Kane et al., 2024, p.11), and the crashes of 737 MAX airplanes (Herkert et al., 2020). Similar kinds of problems obviously occur frequently without gaining broad attention. Furthermore, beyond such direct effects, issues with an SR system may inflict excessive rigidity and regulation on the behavior of humans and of other systems, in an effort to accommodate the system's limitations. Take, for example, the assignment of cars to driving sides and lanes on roads; we would like to avoid enforcing such restrictions enforced in sidewalks and hallways shared by humans and robots as an emergency response to unanticipated problematic reality.

Published discussions of such problems call for early assessment and preemptive technical, economic, and regulatory activities. Early in the days of model-based system engineering (MBSE), France and Rumpe wrote: *It is our view that software engineering is inherently a modeling activity, and that the complexity of software will overwhelm our ability to effectively maintain mental models of a system.* (France and Rumpe, 2007). Over the years, there was great progress in the ability to build executable models. Examples include UML, SysML, Rhapsody, STATEMATE, fUML, xUML, Ptolemy, MATLAB with Simulink and Stateflow, SCADE, UPPAAL, BPMN, Arcadia, Cameo and others; see also list of SySE tools in (Laplante and Kassab, 2022, pp.76,79,208)). In parallel, there were significant advances in applying formal methods to such models (Oliveira et al., 2017; Fremont et al., 2023; de Saqui-Sannes et al., 2021; Zahid et al., 2022; Weyers et al., 2017; Huang et al., 2020; Rahim et al., 2021; Li et al., 2020; Harel et al., 2013). Also, many organizations developed elaborate ad-hoc models to help study the systems from early on in the development process (Lattimore et al., 2022; Gorecki et al., 2019; Lo et al., 2021).

However, despite such advances, ensuring the safety and correctness of complex systems is still a major problem. For example, in a 2024 workshop on safety of autonomous transportation (summarized in (Deshmukh et al., 2024)) many open SySE issues and challenges were discussed, including: (i) incorporating general and domain knowledge in testing and verification; (ii) ensuring that ML training data covers rare but critical scenarios; (iii) exhaustively covering all possible interactions; and, (iv) enhancing usability of formal methods. Similar conclusions about gaps in present methods for early issue identification appear in (Cederbladh et al., 2024; Horváth et al., 2023; Harel et al., 2020; Lee, 2024).

Given that uncovering hidden faults in well-specified or even fully developed systems is still an open problem, it is evident that preemptive fault discovery in super-reactive systems (e.g., extremely complex systems of systems interwoven with their environment) at early development stages poses a major challenge. The added difficulty stems in part from the informal and imprecise nature of requirements in early development stages, from the limited scalability of the tools, and from the reliance on engineers to infer at development time undocumented relations among separately specified requirements. As to the latter concern, these requirements are often specified using different abstractions and a variety of terminologies. Other issues that contribute to the problem include the dependence of verification on a translation to state machines or Petri nets, and the absence of executable and analyzable semantics of certain specification artifacts.

Current AI-based solutions assist in various activities of development, including code generation and debugging, with a prominent example being the GitHub Copilot; see, also, e.g., (AIESE, 2024) and references therein. However, applying such tools in the context of early specification is mostly limited to automated modeling, discussed in Section 3.3.

The roadmap presented here addresses these issues by enabling rigorous execution and analysis subject to domain expertise and world knowledge, and doing so at higher levels of abstraction. The approach uses natural language and relies on the ability of AI-based tools to mimic humans' flexible navigation of complex abstraction relations. It will extend the present use of abstraction, as in object-oriented inheritance relations and counter-example-guided abstraction refinement (CEGAR) in formal verification (Clarke and Veith, 2003; Seipp and Helmert, 2018).

3 THE ROADMAP

In this section, we list the elements of an approach and an architecture for modeling SR systems, including a set of intelligent tools for simulation and analysis, which, together, can enable the much desired early preemptive discovery of hidden faults, while addressing the existing challenges and technology gaps.

In way of bounding the scope of the problem we tackle and hence of our proposed solution, we exclude the use of AI, ML and LLMs in runtime decision making, monitoring, development operations (DevOps), or the formal verification of final code. Moreover, while some reasoning functions of the proposed solution may be similar to those carried out by expert human engineers and domain professionals, we focus on enabling presently impractical or impossible analyses, and much less on automation of manual tasks.

3.1 The Intelligent Development Aide

The *Intelligent Development Aide* (IDA) is a shared layer of services that offers the following to the overall solution: (i) intelligence, including learning, inference, and generative abilities; (ii) NL-based interaction; and, (iii) general world knowledge and certain domain-specific expertise. The IDA will rely on present and future technologies that come under the umbrella of AI, Generative AI, Machine Learning, Deep Learning, Large Language Models (LLMs), etc. It will be constructed, among other things, by fine-tuning, enhancing and extending AI-based tools, relying on techniques like those of (Minaee et al., 2024; Ding et al., 2023; Shani et al., 2023; Tamari et al., 2020; Netz et al., 2024) and future emerging ones. With inputs from specifications of diverse systems, with textual and visual depictions of normal and faulty execution scenarios, the IDA will be trained to recognize unique software and system engineering issues and new delicate kinds of interdependencies.

3.2 The i-model

We introduce a new kind of model, termed *i-model*, which offers fresh perspectives on some common modeling maxims:

First, while precision is commonly needed to ensure correct system implementation, i-models will take advantage of what may appear quite the opposite. They will retain within model entities the expressive power of NL, which includes sensitivity to context, flexible abstraction, generalization, associations, etc. Simulation and analysis tools will then rely on deferred –*just-in-time* (JIT) – interpretation to endow NL and NL-like behavioral specifications with concrete meaning, aligned with the intended context, and abstraction level.

Second, while logical flow and organization are essential to engineering, conceptual abstractions may not always lend themselves to being so depicted. For example, consider the difficulty of modeling a complex network of multiple class inheritances, combined with natural language ambiguity, where, for instance, the word stop could mean a condition of no motion at all, or a process of slowing down to reach that condition, or the action of beginning to press the brake in order to begin this process, etc. In contrast, i-models will accommodate coexistence of multiple, diverse, non-hierarchical, overlapping and dynamic abstraction lattices.

Finally, modularity, encapsulation, and logical decomposition are central principles in engineering in general and in software engineering in particular . However, separately specified requirements often have tacit, unstated dependencies, which show up as exceptions, priorities, alternatives, complementary or concurrent actions, mutually exclusive conditions, etc. It is commonly up to the engineers to infer these implicit relations, and to reflect their understanding in the implementation. In our automated construction of i-models from a wide range of specifications, special focus is put on discovering such unstated relationships and capturing them in the model, despite the entanglement that they may imply.

The i-models will store diverse information, including requirements, goals, behaviors, scenarios and emergent properties, as well as groupings, abstractions, and relations of such entities. It will also contain meta information about potential changes due to the evolution of the system and its environment, allowing further analysis of potential future trajectories.

Finally, the i-model will support *unmodeling* (Marron et al., 2024), i.e., explicitly specifying entities and assumptions that should be excluded or ignored during execution and analysis, as well as operational environments in which the SR system is not expected to operate. Unmodeling will complement capabilities of existing modeling techniques to specify the exact intended operational design domain (ODD), directing IDA-based tools where to apply their vast and important knowledge and where not to.

The immense knowledge stored in the i-model will be divided among three realms: (i) the entities themselves, including structured data and unstructured NL documents; (ii) the relationships between entities, represented in the i-model database; and, (iii) the general and application-specific knowledge captured in the IDA components, both in advance, and following the building and analysis of a given i-model.

3.3 The i-model Builder

Inputs to i-model building will include: requirements documents, specifications of reusable components (Benveniste et al., 2018), manual risk analyses (Bjerga et al., 2016; Haimes, 2018), entire models in various modeling languages, program code, documentation, example run logs of early prototypes, test cases, etc. Additional information, corrections and guidance provided interactively by engineers during model building will also be retained. Furthermore, the i-model builder can initiate queries, asking the engineers to supply missing information or to confirm intermediate engine inferences. For example, when preparing a model for simulating complex traffic scenarios in a busy intersection, the system may remind the domain experts to include various combinations of weather conditions, and road surface states.

Beyond the now increasingly common translation of NL specifications into basic object models and computer programs, a unique feature of the i-model builder will be the automated, and optionally interactive, discovery and recording of undocumented tacit interdependencies among separately specified entities. For example, consider separately specified rules for an autonomous vehicle (AV), which may cause the AV to accelerate, as when entering a highway or when instructed to follow another vehicle, or when returning to normal speed after a temporary slow down; consider a second set of rules specifying maximum legal speed and maximum recommended speed under certain conditions. The fact that the rules in the second set constrain or may be in conflict with rules in the first set, will be automatically captured and explicitly specified at model-building time. When the effect is clear, e.g., that one rule takes priority over the other, this explicit specification will be generated automatically up front (where today it may be left as an implementation detail). When the relation is in question – say, what to do if the leader of a convoy exceeds the legal speed limit – the i-model builder will consult the engineers.

The builder will provide succinct summaries of the input information and elaborate on its inferences, applying logic and domain knowledge. The added information will also be stored in the i-model. For example, in our experiments with an LLM in creating a model from the description of a traffic scenario, the LLM added pedestrian objects, which were absent from the original requirements.

The i-model will use a rich modern database to store the structured and unstructured information and the associated connections and relationships. As this structure will likely be too complex for humans to navigate and maintain directly, it will be supported by a multi-view presentation (see, e.g., *slices* in (Nejati et al., 2012)), *projecting* requested information as stand alone succinct text descriptions and dynamically created diagrams (e.g., Statecharts). Since many entities will already be in NL, the projection itself will be intuitive, but will still require other components and human review for confirmed interpretation (see also Section 4). Once confirmed, such projections can also serve as concise memory, to be fed back to the i-model.

We expect the construction of the i-model builder to extend the growing amount of research on using GAI and LLMs for creating models from NL inputs; see, e.g. (Ramackers et al., 2021; Netz et al., 2024; Yaacov et al., 2024; Harel. et al., 2024; Cosler et al., 2023; Kourani et al., 2024).

3.4 The i-model Execution and Simulation Engine

This component performs dynamic, deferred, just-in-time (JIT) interpretation of the definite and the NL information in the i-model, and outputs descriptions of possible runs and the unfolding of "what if" scenarios at desired abstraction levels, as narratives, event logs, and even synthetic videos.

For illustration, consider the following example, which we ran on OpenAI ChatGPT. Prompt (slightly edited): *"Consider a 3 lane street. 30 equal-size passenger cars are filling the street, standing in front of a traffic light. No cars arrive from the back of the queue* *throughout this scenario. The third car in the middle lane stalls and cannot move. No one stops to help. Describe the behaviors and positions of the 30 cars after the light turns green. Assume the light is green for a very long time."*

Here are a few excerpts from the LLM's response: *"When the traffic light turns green:...**1. Cars in the Middle Lane...:** (a) The first two cars in the middle lane (Cars 1 and 2) will start moving forward as the light turns green. (b) The third car in the middle lane (Car 3) will not move...(c) The cars behind the third car...will initially wait, expecting the third car to move. After realizing that it is not moving, they will either: (i) Start honking,...; (ii) Some drivers may try to change lanes...(iii) Some drivers may remain stuck behind the third car if changing lanes is not possible... **2. Cars in the Left Lane...and Right Lane...:** (a) The cars in these lanes...will begin moving forward... (b) The first few cars...positioned parallel to the stationary third car, will drive past it without much issue. (c) The cars behind them will continue moving forward, potentially picking up speed..."*

When we divided the road into small virtual cells, the LLM was able to describe car movements through the numbered cells, making the scenario more mathematically analyzable.

Note how the LLM incorporated into the responses its domain knowledge; for example, adding the possibilities of changing lanes and honking, which were not in the original specification.

The LLM responses also contained errors; for example, they suggested that cars blocked behind the stalled car may not be able to change lanes at all, ignoring the specification that the traffic light stays green indefinitely, and no new cars arrive during the scenario. In Section 4 we discuss approaches for dealing with such issues.

This example and others show that LLMs can produce execution logs, both in structured form and as a continuous narrative, which can then be checked using a variety of techniques.

The execution engine will, of course, benefit from state-of-the-art execution and simulation techniques, like those in SysML, Statecharts, and Scenario-based programming, or in direct execution of NL specifications, as in (Tamari et al., 2020) and references therein. Borrowing from techniques for test-case generation (Wang et al., 2024), the execution engine will also automatically generate and process batches of diverse, yet relevant, "what if" scenarios, and store their execution results for further processing.

3.5 The i-model Analysis Engine

The i-model analyzer will carry out the equivalent of formal model-checking, searching – proactively – for execution trajectories that lead to fault states. Treating the model as an NL-enriched graph, it will traverse its paths, interpreting entities and relationships subject to general and domain knowledge, including causalities, interdependencies and risks, while abiding by *unmodeling* – explicit specifications of what to exclude.

For example, assume that the model includes rules like "a vehicle should never proceed into an intersection when the traffic light is red", and "drivers and vehicles should always obey police person's directives'. NL processing combined with domain knowledge can equate terms that appear in the model, like "go", "proceed", "drive forward", and others, into a single concept, and can categorize the possible directives of a police person into categories like "go", "stop", "turn", etc. During simulation of actual scenarios, the behavior of the vehicle, e.g., changing location coordinates, can then be described in words and associated with the recognized terminology. The system can then detect when a vehicle's behavior violates such rules. Furthermore, a model checker or an SMT constraint solver may then be able to detect that there is a potential conflict between these two rules, which would require prioritization or some other means of resolution.

In addition, the analyzer will offer query capabilities, e.g., for investigating complex scenarios, or the many connections of a given entity. It will also interface with classical model checkers and satisfiability modulo theory (SMT) constraint solvers for inspecting well-structured subsets and projections of the i-model, and for presenting the answers back in NL.

For example, we described to ChatGPT two parallel synchronous state machines. With some trial and error, aided by checking and corrections by engineers, the LLM was able to say whether certain composite states were reachable or not, describe relevant paths, and construct the full state graph of the composite machine with all composite states and transitions.

Other analytic LLM capabilities that can support i-model analysis are described in papers like (Harel. et al., 2024; Sultan and Apvrille, 2024). These include computing when two independent periodic events may occur simultaneously, explaining behavior, articulating system properties, checking model consistency, etc. Furthermore, it is expected that LLM general and domain-specific analytical capabilities will be extended and deepened, and they are likely to be intertwined with ongoing research in software and system engineering. Developments along the present roadmap can incorporate such enhancements, and target them specifically at early fault-detection.

Still, automated validation techniques must continue to be researched and developed. Even if our fault-detection solution is found to work well on reasonably tractable models, like compositions of small specifications, can one trust the solution's answers for larger problems? And wouldn't a trusted automated external validation tool make the AI-based solution actually unnecessary? We believe that with a combination of well-documented abstraction relations, AI-explainability, randomized testing of model answers, and powerful projection of relevant model perspectives, one can create high confidence in the model's answers. See Section 4 for further details.

3.6 Emergent Effect Detector

This component accepts outputs of system simulations, looking for expected and unexpected patterns and emergent effects, both structural/spatial and behavioral/temporal. Such effects may be previously specified as desired, undesired, or perhaps acceptable, or they may require assessment.

The tool will rely on the immense body of work in recognizing patterns, emergent effects, anomalies, etc., in formally organized sequential data, like discrete event logs or continuous signals, and in spatial and structural information, like images and videos. See, e.g., (Pang et al., 2021; Fieguth, 2022; Noering et al., 2021; Bartocci et al., 2022). The results will be presented formally and in NL for manual and automated analysis.

3.7 Repair Advisor

The i-model's sheer size may interfere with its maintenance, calling for a repair advisor that accepts a description of an issue and proposes changes to the system or to its technical and physical environment. A key distinction from common program repair (Zhang et al., 2023) is the primary focus on pinpointing the model components that should be changed and on describing the ensuing impact on system behavior, while the technical details of the actual change are secondary.

4 DEALING WITH LLM SHORTCOMINGS

We offer the following methodological principles in order to counteract known weaknesses in AI- and LLM-based techniques, justifying their inclusion in a foundation of a robust engineering tool. The weaknesses include the possibility of faulty inference and "hallucination", scalability issues, and vulnerability to various attacks:

Abstraction. In early stages of development, many aspects of the specification are aggregated in high-level abstractions, which by their very nature reduce complexity and the magnitude of the state space. In addition, at any stage, when the available knowledge is cluttered by excessive amounts of detail, stakeholders can raise the level of abstraction in the available specification to achieve the necessary ad-hoc, temporary simplification.

Modularity. Designers can limit the scope of the challenges delegated to AI and LLM techniques by dealing with encapsulated components, and abstracting each component's view of the behavior of the rest of the system and the environment.

Human Review. Recall that in a classical development process, any failure in testing or formal verification is subject to human review: Is the problem real? Was there a problem in the definition of execution of the test and the verification? Can we recreate the issue? etc.

Indeed, at all development stages, from initial requirement elicitation to advanced sprints in agile development, domain experts and engineers may raise "what if" questions, and point at issues, some of which may be irrelevant due to misunderstandings, mismatching assumptions, or simply the forgetting of already-specified elements. The team, including the person raising the issue, then check if the issue at hand is indeed one that should be fixed, whether documentation of other details and assumptions should be improved, whether the issue can be dismissed by a succinct answer, etc.

Explainable AI. Applying state of the art explainability tools to the IDA observations can assist in dismissing erroneous or superfluous ones, and focusing on relevant ones. This process can also help in enhancing both the IDA and the system i-model at hand to improve the overall quality of the automated engineering process.

For example assume that in observing real world or simulated behavior of an AV being developed, the IDA reports an unsafe, unexpected slowing down in the middle of a highway. Applying explainable AI to the vehicle's logic may supply the reason, such as its having detected a pothole in the road. This can then be translated into simply dismissing the observation, or improving the observation abilities of the IDA, or in case of a false detection by the vehicle, fixing its sensors and the associated logic.

Standard Security Procedures. The vulnerabilities of neural networks to various kinds of attacks, in training and in adversarial use, should be addressed with standard practices for this well studied area, including data controls, access controls, diverse monitoring, etc.
Domain Specific Training. The LLMs and other AI techniques involved in the proposed solution should be further trained and updated with domain-specific knowledge, and with extensive background about software engineering practices. Such training should cover also the specific accumulated experience in using the new architecture and methods.
Redundancy. Especially when safety is an issue, important decisions, tests and validation should be done by several tools that rely on different resources and designs. Furthermore, the decisions of AI based systems may be guarded by safety rules programmed from more classical specifications (Harel et al., 2024).
Training Data Filtering and Curation. Special care should be taken to ensure that the IDA is trained on valid, clean data. For example, the IDA should not learn from malicious inputs, and when learning from valid systems and processes, it should not violate proprietary rights associated with such sources.

5 CONCLUSION

We are currently in the process of initiating a research project following the roadmap presented here. Development of models and tools that enable simulation and analysis of highly complex systems based only on early specifications can dramatically enhance our ability to develop reliable, safe, and productive super-reactive systems. A combination of the recent advances in AI and a fresh perspective on what may and may not qualify as a model entity, or be acceptable as a simulation result, may enable the achievement of this tantalizing goal.

ACKNOWLEDGMENTS

This research was funded in part by an NSFC-ISF grant to DH, issued jointly by the National Natural Science Foundation of China (NSFC) and the Israel Science Foundation (ISF grant 3698/21). Additional support was provided by a research grant to DH from Louis J. Lavigne and Nancy Rothman, the Carter Chapman Shreve Family Foundation, Dr. and Mrs. Donald Rivin, and the Estate of Smigel Trust.

REFERENCES

AIESE (2024). 15th Int. Conf. on AI-empowered Software Engineering – AIESE 2024 (Formerly JCKBSE). https://easyconferences.eu/aiese2024/; Accessed Aug. 2024.

Bartocci, E., Mateis, C., Nesterini, E., and Nickovic, D. (2022). Survey on mining signal temporal logic specifications. *Information and Computation*, 289:104957.

Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T. A., Larsen, K. G., et al. (2018). Contracts for system design. *Foundations and Trends® in Electronic Design Automation*, 12(2-3):124–400.

Bjerga, T., Aven, T., and Zio, E. (2016). Uncertainty treatment in risk analysis of complex systems: The cases of stamp and fram. *Reliability Engineering & System Safety*, 156:203–209.

Cederbladh, J., Cicchetti, A., and Suryadevara, J. (2024). Early validation and verification of system behaviour in model-based systems engineering: a systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(3):1–67.

Clarke, E. and Veith, H. (2003). *Counterexamples revisited: Principles, algorithms, applications*. Springer.

Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., and Trippel, C. (2023). nl2spec: interactively translating unstructured natural language to temporal logics with large language models. In *CAV*, pages 383–396. Springer.

de Saqui-Sannes, P., Apvrille, L., and Vingerhoeds, R. (2021). Checking SysML models against safety and security properties. *Journal of Aerospace Information Systems*, 18(12):906–918.

Deshmukh, J., Könighofer, B., Ničković, D., and Cano, F. (2024). Safety Assurance for Autonomous Mobility (Dagstuhl Seminar 24071). *Dagstuhl Reports*, 14(2):95–119.

Ding, N., Qin, Y., Yang, G., Wei, F., Yang, Z., Su, Y., Hu, S., Chen, Y., Chan, C.-M., Chen, W., et al. (2023). Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Mach. Intel.*, 5(3):220–235.

Fieguth, P. (2022). *An introduction to pattern recognition and machine learning*. Springer.

France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE'07)*, pages 37–54. IEEE.

Fremont, D. J., Kim, E., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A. L., and Seshia, S. A. (2023). Scenic: A language for scenario specification and data generation. *Machine Learning*, 112(10).

George, A. S. (2024). When trust fails: Examining systemic risk in the digital economy from the 2024 CrowdStrike outage. *Partners Universal Multidisciplinary Research Journal*, 1(2):134–152.

Gorecki, S., Ribault, J., Zacharewicz, G., Ducq, Y., and Perry, N. (2019). Risk management and distributed simulation in papyrus tool for decision making in industrial context. *Comput. & Indus. Engineering*, 137.

Haimes, Y. Y. (2018). Risk modeling of interdependent complex systems of systems: Theory and practice. *Risk Analysis*, 38(1):84–98.

Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., and Weiss, G. (2013). On composing and proving the correctness of reactive behavior. In *EMSOFT 2013*, pages 1–10. IEEE.

Harel., D., Katz., G., Marron., A., and Szekely., S. (2024). On augmenting scenario-based modeling with generative AI. In *MODELSWARD 2024*, pages 235–246.

Harel, D., Marron, A., and Sifakis, J. (2020). Autonomics: In search of a foundation for next-generation autonomous systems. *Proceedings of the National Academy of Sciences*, 117(30):17491–17498.

Harel, D. and Pnueli, A. (1984). On the development of reactive systems. In *Logics and models of concur. sys.*, pages 477–498. Springer.

Harel, D., Yerushalmi, R., Marron, A., and Elyasaf, A. (2024). Categorizing methods for integrating machine learning with executable specifications. *Science China Information Sciences*, 67(1):111101.

Herkert, J., Borenstein, J., and Miller, K. (2020). The Boeing 737 MAX: Lessons for engineering ethics. *Science and engineering ethics*, 26:2957–2974.

Horváth, B., Molnár, V., Graics, B., Hajdu, Á., Ráth, I., Horváth, Á., Karban, R., Trancho, G., and Micskei, Z. (2023). Pragmatic verification and validation of industrial executable SysML models. *Systems Engineering*, 26(6):693–714.

Huang, E., McGinnis, L. F., and Mitchell, S. W. (2020). Verifying SysML activity diagrams using formal transformation to petri nets. *Systems Engineering*, 23(1):118–135.

Kane, B. R., Webber, S., Tucker, K. H., Wallace, S., Chang, J., Mccarthy, D., Murphy, D., Egel, D., and Wingfield, T. (2024). Threats to critical infrastructure. *Rand Corporation Research Reports*.

Koopman, P. (2024). Anatomy of a robotaxi crash: Lessons from the Cruise pedestrian dragging mishap. *arXiv preprint arXiv:2402.06046*.

Kourani, H., Berti, A., Schuster, D., and van der Aalst, W. M. (2024). Process modeling with large language models. In *International Conference on Business Process Modeling, Development and Support*, pages 229–244. Springer.

Laplante, P. A. and Kassab, M. (2022). *What every engineer should know about software engineering*. CRC Press.

Lattimore, M., Karban, R., Gomez, M. P., Bovre, E., and Reeves, G. E. (2022). A model-based approach for Europa lander mission concept exploration. In *2022 IEEE Aerospace Conference (AERO)*, pages 1–13. IEEE.

Lee, E. A. (2024). Certainty or intelligence: Pick one! In *Design, Automation & Test in Europe (DATE)*, pages 1–2. IEEE.

Li, N., Tsigkanos, C., Jin, Z., Hu, Z., and Ghezzi, C. (2020). Early validation of cyber–physical space systems via multi-concerns integration. *Journal of Systems and Software*, 170:110742.

Lo, C., Chen, C.-H., and Zhong, R. Y. (2021). A review of digital twin in product design and development. *Adv. Eng. Informatics*, 48:101297.

Marron, A., Cohen, I. R., Frankel, G., Harel, D., and Szekely, S. (2024). Challenges in modeling and unmodeling complex reactive systems: Interaction networks, reaction to emergent effects, reactive rule composition, and multiple time scales. *Springer CCIS*.

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., and Gao, J. (2024). Large language models: A survey. *arXiv preprint arXiv:2402.06196*.

Nejati, S., Sabetzadeh, M., Falessi, D., Briand, L., and Coq, T. (2012). A SysML-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. *Information and Software Technology*, 54(6):569–590.

Netz, L., Michael, J., and Rumpe, B. (2024). From natural language to web applications: Using large language models for model-driven software engineering. In *Modellierung 2024*, pages 179–195. Gesellschaft für Informatik eV.

Noering, F. K.-D., Schroeder, Y., Jonas, K., and Klawonn, F. (2021). Pattern discovery in time series using autoencoder in comparison to nonlearning approaches. *Integrated Computer-Aided Engineering*, 28(3).

Oliveira, R., Palanque, P., Weyers, B., Bowen, J., and Dix, A. (2017). State of the art on formal methods for interactive systems. *The handbook of formal methods in human-computer interaction*, pages 3–55.

Pang, G., Shen, C., Cao, L., and Hengel, A. V. D. (2021). Deep learning for anomaly detection: A review. *ACM comput. surv.*, 54(2):1–38.

Rahim, M., Boukala-Ioualalen, M., and Hammad, A. (2021). Hierarchical colored Petri nets for the verification of SysML designs-activity-based slicing approach. In *Advances in Computing Systems and Applications: Proc. 4th Conf. on Comp. Sys. and App.*, pages 131–142. Springer.

Ramackers, G. J., Griffioen, P. P., Schouten, M. B., and Chaudron, M. R. (2021). From prose to prototype: synthesising executable UML models from natural language. In *MODELS-C*, pages 380–389. IEEE.

Seipp, J. and Helmert, M. (2018). Counterexample-guided cartesian abstraction refinement for classical planning. *J. of Artificial Intel. Res.*, 62.

Shani, C., Vreeken, J., and Shahaf, D. (2023). Towards concept-aware large language models. *arXiv preprint arXiv:2311.01866*.

Sultan, B. and Apvrille, L. (2024). Ai-driven consistency of sysml diagrams. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, pages 149–159.

Tamari, R., Shani, C., Hope, T., Petruck, M. R. L., Abend, O., and Shahaf, D. (2020). Language (re)modelling: Towards embodied language understanding. In Jurafsky, D., Chai, J., Schluter, N., and Tetreault, J., editors, *Proc. of the 58th Annual Meeting of the ACL*. ACL.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.

Weyers, B., Bowen, J., Dix, A., and Palanque, P. (2017). *The handbook of formal methods in human-computer interaction*. Springer.

Yaacov, T., Elyasaf, A., and Weiss, G. (2024). Boosting LLM-Based Software Generation by Aligning Code with Requirements. In *Proc. 14th Int. Model-Driven Requirements Engineering Workshop (MoDRE)*.

Zahid, F., Tanveer, A., Kuo, M. M., and Sinha, R. (2022). A systematic mapping of semi-formal and formal methods in requirements engineering of industrial cyber-physical systems. *J. of Intel. Mfg.*, 33(6).

Zhang, Q., Fang, C., Ma, Y., Sun, W., and Chen, Z. (2023). A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–69.