On using large language models pre-trained on digital twins as oracles to foster the use of formal methods in practice

Serge Autexier[0000−0002−0769−0732]

German Research Centre for Artificial Intelligence (DFKI), Enrique-Schmidt-Str.5, 28359 Bremen serge.autexier@dfki.de
https://www.dfki.de/˜serge

Abstract. Formal methods based on formal logical or mathematical symbolic techniques provide the highest standards to analyse and ensure safety and security properties of cyber-physical systems—but require a large overhead to specify and especially to verify system properties. The laborious and often manual and creative tasks consist of coming up with the appropriate specifications, to specify required lemmas or to guide the verification process. Once properties, lemmas, proof guidance or a proof itself is given, checking it rigorously for correctness is an easy task. Large Language Models (LLMs) have demonstrated remarkable proficiency in a variety of domains, and there is a recent trend of research that tries to leverage that potential for the creative parts of formal verification. On the other hand, digital twins as virtual replicas of physical systems or processes are used to simulate and analyse real-world systems or scenarios, allowing for predictive maintenance, optimization, and testing of the systems. They can be used to predict and prevent failures, optimize processes, and test different designs in a virtual environment. They can also aid monitoring of real systems and offer support to adapt systems to new situations, e.g., in case of failures to simulate how to reestablish safe operations. Starting from examples of research results using LLMs for symbolic techniques, we advocate researching how to systematically design digital twins for systems under investigation, use simulation capabilities to explore system behaviours and assess their formal properties to create high-quality training data sets to pre-trained LLMs offline. Such LLMs shall then serve in specific verification tasks in practice, to take care of the difficult and, so far, manual task of generating appropriate candidate properties, lemmas or proofs that are subsequently automatically and efficiently checked for correctness using formal techniques.

Keywords: Formal Methods · Large Language Models · Digital Twins

1 Introduction

Formal methods provide the highest reliability that systems comply with their safety and security requirements. However, the effort required to apply them in practice is extremely high and thus far they are only used if required, e.g., for the

higher safety integrity levels (SILs) of the IEC 61508 standard [15] (and related) or when costs of failures are extremely high. Compared to the less reliable but much adopted testing of software, formal methods require the formalization of a system in development or under investigation in an appropriate formal logical or mathematical language, the specification of the safety and security requirements and subsequently the verification of the properties on the system specification. Verification is more often than not an interactive manual process, either requiring the experts guiding the proofs or decomposing the properties into sub-properties and lemmata. And even if automatic, the verification process may suffer from state space explosion. This makes the use of formal methods laborious in practice already during the development phase. When it comes to the deployment phase system failure may nevertheless occur, e.g., due to failing or unreachable sub-systems or successful security attacks, using formal methods to design the reactions to bring the system back towards a safe and secure operation is nearly excluded due to lack of time.

Large Language Models (LLMs, e.g., [7, 3] have demonstrated remarkable proficiency in a variety of domains and there is recent interest to explore their potential as oracles in formal methods (See Section 2), though focusing mainly on the development phase of systems and not deployment phase where system are in operation.

First presented in 2002 by Grieves [13], digital twins have gained a lot of interest in industry as digital counterparts to the physical system or processes, that can be used to for simulation, integration, testing, monitoring as well as maintenance planning. Being a concept that covers the whole product life cycle, they thus address both the development phase as well as the deployment and maintenance phases.

This paper proposes leveraging LLMs as oracles to be used for the different tasks that make the use of formal methods difficult in practice. To do so, it proposes to systematically organise the development of domain-specific LLMs in combination with formal methods for the different specific tasks, and especially also exploit digital twins to develop domain-specific LLMs to assist during the deployment and maintenance phase.

The paper starts with a review of examples using LLMs for formal methods in Section 2. Section 3 provides an overview of the use of digital twins and especially for which specific tasks which arise in the product life cycle they are used for. Based on these Section 4 will discuss the role LLMs shall / could play to foster the use of formal methods for specific tasks throughout the whole software development life cycle (SDLC), which includes the deployment and maintenance phases where digital twins are used. For the different cases, we will discuss the role and requirements for formal methods techniques that arise from their interplay with LLMs, before a summarizing discussion in Section 5 concludes the paper.

2 LLMs for formal methods

Reviewing methodology. The following is a review of existing work on using LLMs for formal methods. It is not a systematic review in the proper sense, but search has been conducted through usual internet search engines (Google[1], Startpage[2]) using combinations "large language model" and "foundation model" and shortcuts "LLM" in combination with the formal software development methods of interest such as "(automated) theorem proving", "logical formalization", "program verification", "program synthesis" and similar. Futhermore, pointers given by authors of found literature to interesting related where also followed.

2.1 LLMs in theorem proving

Proving formal theorems is a core activity in formal methods. Automated theorem proving and model checking procedures become more and more powerful but depend on having the right properties and lemmata formulated. We will come back to this in the next Section 2.2. The painful part of applying formal methods arise when proofs have to be constructed interactively with an interactive proof assistant such as Coq [25], Isabelle [33], or Lean [6] to name a few.

Starting from the LLama framework [30] Azerbayev et al. [1] pre-train the LLM LLemma on Proof-Pile-2, which is mixture of scientific papers (29 billion tokens), web data containing mathematics (15 billion tokens), and mathematical code (11 billion tokens). Given a mathematical problem formulation, LLemma generates a proof as typically found in textbooks with a mix of natural language and formulas. Furthermore, it is able to generate proof scripts for the Isabelle proof assistant in the declarative Isar style, if, additionally, a formal statement of the problem is provided. Furthermore, with a formal statement of the problem and a proof state in Lean, it generates the subsequent steps. All details can be found in [1].

Similarly, the Baldur system of First et al. [8] also targets the Isabelle proof assistant and, given a formalized problem statement, generates a proof script, which can then be checked by Isabelle. The interesting addition compared to LLemma is that in case Isabelle fails to check the generated proof script, it takes the generated error message together with the problem statement and last generated candidate proof as input to generate a new, adjusted proof script.

As a final example in that line of research Song et al. [28] provide an LLM as copilot to interactively create proofs in Lean. Analogously to code copilots, the user builds the proof script interactively and obtains on demand suggestions of proof steps (tactics), completes intermediate proof goals by automatic proof search, or helps selecting relevant premises. To this end specific commands have been included in the proof script language: these trigger the corresponding LLM assisted process and suggestions are provided on the side, from which the user can then copy and paste.

1 www.google.com
2 www.startpage.com

2.2 LLM for property formalization

Another challenge in using formal methods is that requirements which are typically stated in natural language need to be translated into the formal language at hand. In the context of the Isabelle proof assistant Wu et al. [35] studied how LLMs can generate formalizations of mathematical problems, a process denoted as autoformalization [32, 29]. Wu et al. pre-trained a version of GPT-2 on two corpora of data: The first containing 12500 middle school and high school mathematical competition problems in seven mathematical categories and second a benchmark of 488 mathematical competition statements manually formalized by humans in three different formal languages. To evaluate the autoformalization they iteratively trained the neural theorem prover LISA [17] on the generated formalization and proofs and measured if the performance of LISA is improved, i.e., that the neural prover benefits from proving the generated theorems.

2.3 LLM in program verification

Program verification is a core discipline in formal methods and decades of research have been going into this. The starting point is actually to come up for a given program in some programming language with a formal specification that can be annotated on the code, on which basis the verification happens. This is related to the previous section in the sense that formalizations need to be found, but more tailored to actual programs and properties about programs. Here, formal annotations need to be created, which often is a manual process and also requires some creativity, e.g., in the case of coming up with invariants. There exists a variety of automated program specification generation tools which typically rely on templates defined by experts. They are used to generate candidate specifications, which are then filtered by verifiers to eliminate incorrect specifications.

Ma et al. [22] consider java programs and JML annotations to use an LLM to overcome the pre-definition of templates. Using prompt engineering, they took advantage of OpenAI's `gpt-3.5-turbo-1106` as LLM to create a conversation between the LLM and the JML specification verifier OpenJML [5], where the LLM proposes specifications for a program and the verifier either accepts or provides feedback why the specification is rejected. This is comparable to the approach used in [8] for generating Isabelle proof scripts.

Pei et al. [26] tackled the problem of creating invariants for programs: they also chose Java programs as application domain and created a dataset of programs with invariants at different relevant positions in the program source code. From that they pre-trained an LLM (GPT-4) on the source codes annotated with invariants. The pre-trained LLM in turn was then fine-tuned to take as input the source code of a program, and a target program position, to output a list of invariants for the designated program position.

Inspired by Pei et al. [26], Wu et al. [34] proposed the LEMUR framework as fully automated framework combining LLMs and reasoners. In this framework,

LLMs propose program invariants, which are then checked by automated reasoners. They present LEMUR as a proof system for which they prove soundness and termination. They instantiate the framework with OpenAIs GPT-3 and GPT-4 as starting LLMs. Similarly to First et al. [8] they use pre-trained LLMs to generate invariants as well as to correct invalid / non-provable invariants based on the feedback of the underlying reasoner. As verification tools they used ESBMC from Gadelha et al. [10] and UAUTOMIZER from Heizmann et al. [14].

2.4 LLMs in program synthesis

LLMs in the style of CoPilots are increasingly used to help writing programs. While these start from natural language descriptions, there is no guarantee that the generated programs comply with the original specification or are free of flaws. Investigations have been conducted in that direction, where LLMs are given formal specifications to generate programs that fulfil that specification.

The best-performing synthesizers in the domain of formal synthesis with precise logical specifications are still based on enumerative algorithms (see Li et al. [20]). Li et al. investigated the use of LLMs in that enumeration process. They propose two variants how the enumeration process and the LLMs are cooperating. They start from GPT-3.5-turbo and perform prompt engineering using formal property specifications in SMT-lib format and prompt directives.

Another line research, actually driven by the motivation to control the hallucinations of LLMs, is presented by Jha et al. [16]. In that work an overall approach similar to First et al. [8] and Ma et al. [22] is used, where an LLM is used to propose solutions, which are then assessed by a verifier based on a given formal specification: if the proposed solution is correct, the solution is returned as a verified solution. If the solution has flaws, counter-examples are added to the query which was re-submitted to the LLM in order to generate a new solution. They tested this approach in the AI planning domain [11], using ChatGPT as LLM, which is asked to generate AI plans for stated planning problems. The generated plans are verified by plan execution to check if the generated plan indeed solves the problem; invalid intermediate plans are added to the query indicated as being invalid plans in order to discard them to be re-proposed.

3 Digital twins

Digital twins have gained a lot of interest in industry as digital counterparts to the physical system or processes, that can be used to for simulation, integration, testing, monitoring as well as maintenance planning. Digital twins are also used in the different phases of the life cycles of systems and processes (cf. Negri et al. [24]), similar to those life cycle phases considered to structured this paper. In an effort introduce more structure Kritzinger et al. [19] proposed a refined terminology[3] of digital twins according to which automatic data flows exists

[3] Kritzinger et al. [19] denote the counterparts as 'objects', i.e., 'Real Object' and 'Digital Object', as they also deal with processes. We used system here as we are only concerned with software and hardware.

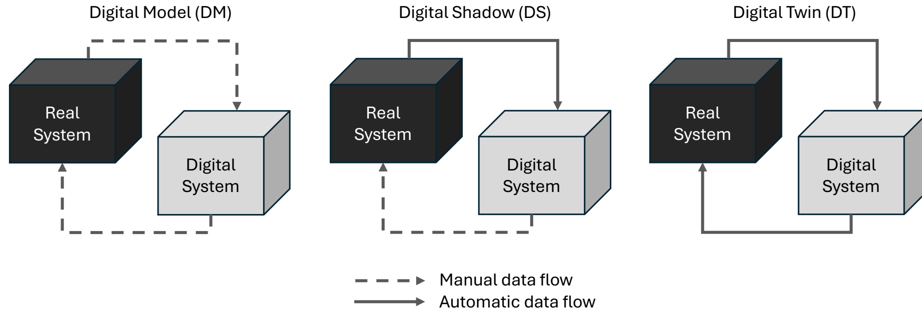

Fig. 1. Terminology for digital twins following Kritzinger et al. [19]

between the twin and the real system under consideration (cf. Fig. 1): They propose to denote as (i) *Digital Models (DM)* those systems, where there is neither automatic data flow from the real system to the digital system nor vice versa; (ii) *Digital Shadow (DS)* if there is only automatic data flow from real system to the digital system, and (iii) *Digital Twin (DT)* if data automatically flows from either system to the other.

In their review, which also included processes and not only systems, from2018 Kritzinger et al. [19] noted that most works were concerned with Digital Shadows (35 percent), followed by Digital Models (28 percent) and actually only 18 percent considered Digital Twins in the sense of their definition.

While there is still a debate[4] around that terminology which seems to revolve around the fact that Kritzinger et al. [19] assign digital systems exclusively to be either a model, a shadow, or a twin, which is not necessarily. Indeed, a digital model used a design time, may evolve into a digital shadow or digital twin for a real system, while a copy may still serve to model next versions of the system. Similarly, digital shadows may serve for simulations how to adjust configurations in the physical system. Also a digital twin may be taken partially offline to be a digital shadow in order to serve for the aforementioned types of simulations. In the following and especially in Section 4, we will employ that terminology in order to make explicit the type of interaction between physical and digital systems under consideration.

Löcklin et al. [21] conducted in 2020 an overview of the use of digital twins specifically for verification and validations in industrial automation systems. Their findings are that in that domain, digital twins are used for one or more of the three following tasks: exploratory investigation, testing and formal proving. An investigation on 33 approaches using digital twins for verification and validation showed that 13 are used for exploratory investigations, 17 for testing and only 3 for formal proving.

[4] https://www.linkedin.com/pulse/digital-model-shadow-twin-michael-grieves, published: April 6, 2023, accessed: August 22, 2024

All 33 approaches make use of digital models of physical systems, but only 19 exploit the simulation capabilities in the sense of *Digital Models (DM)*. 15 combine available operational data and data analysis on the *Digital Shadow (DS)* in a central interface. Surprisingly, none made use of the possibility to synchronize the physical system and the digital system, i.e., making a *Digital Twin*, while in principle having an up-to-date model would be beneficial to develop for automated, dynamic verification and validation during operation, a key feature of intelligent autonomous systems to safely self-adapt during operation.

Regarding the specific tasks the models are used for, the authors could derive 3 types of activities: first, digital twins are used for **what-if-analysis**, for evaluating different options, or effect of bringing back to normal operation specific parts (21 approaches). Second, 5 approaches used them for **cross-validation** to detect differences between models and the behaviour of real systems. Thirdly, the digital twin is used for **evidence advice** (7 approaches), to measure test coverage or respectively to find gaps in test coverage.

4 Fostering further practical use of formal methods

In this section we discuss along the life cycle of a system the different activities that need to be performed using formal methods and which shall be supported by specifically trained LLMs. The guiding idea is that checking with formal methods is much simpler than searching, the goal is that LLMs take over the tasks of proposing a solution, that can then be checked using formal methods. This is very much like in theorem proving where searching for an appropriate instance of an existentially quantified variable is the difficult part and checking if a proposed instance satisfies the formula is simpler—if the proposed instance is correct. Analogously, output of the LLMs, even if trained on excellent data, may be wrong, which is only detected through the use of formal methods techniques. To serve the practical use of formal methods, the roles played by the formal methods need to be taken into account: first, they shall be able to quickly check if a solution is correct and if not, rather than performing time consuming search, provide feedback that may be used by the LLMs or the developer to adjust the solution or the system. Of course, powerful formal methods proof search and checking techniques are still required, but may be more used during the training time in order to generate and check suitable data, on which the LLMs are pre-trained.

4.1 Requirements Elicitation

Challenge using formal methods. Requirements for systems under development are formulated in natural language and a difficult task is to translate these requirements into formal specifications.

Support offered by LLMs. Large language models can be used to analyse natural language descriptions of safety and security requirements (see Rajkovic and

Enoiu [27]) and generate formal specifications in a formal language such as mathematical logic or model-based specifications. This can help to reduce the complexity of the requirements gathering process, the challenge being that the LLMs produce results ensuring that the requirements are unambiguous and consistent. This was a reason to introduce the Attempto Controlled English [9] and the challenge is to design means to create datasets and train LLMs that support the task, while accommodating a more flexible language as well as also other languages than English.

The work presented in Section 2.2 are first steps in that direction, but there is still a lot of room for improvement. The work in Section 2.2 targeted the mathematical problem domain and it seems sensible to consider other domains that pertain to typical application domains of formal methods.

Requirements for formal methods. The method presented in the example of Section 2.2 to check the usefulness of autoformalization is based on whether the neural theorem prover can be improved based on the additional knowledge. This is possible and valid regarding the application domain of mathematical competition problem, but not for the task of providing formalizations for program verification. For that one will have to develop means how to check created formalization and maybe also think about solutions like autoencoders, to recreate the original natural language specifications from the formalization. Another possibility could be to include humans in the loop to visualize the behaviour induced by the requirements. If the requirements would allow to create a digital twin (in the sense of a *digital model*), which could be used for simulations, would be interesting. This could be a possible topic to investigate, by not going the full way to program synthesis (see Section 2.4), but actually generating digital models. Alternatively, properties that can be derived from the generated requirements could be generated and provided to the user to allow the user to check these and assess if this is indeed what he/she wanted to specify.

Training data for pre-training LLMs. In order to train LLMs, data from past and ongoing developments could be used. The challenges will probably consist of getting access to such data, as they are usually not publicly available and also may contain sensible information. This may require thinking in the direction to protect the data by using approaches like federated learning with dynamic federations of participating institutes and industry.

4.2 Formal Specification

Challenge using formal methods. To specify a system requires to describe its environments, the input and the output and the relation between these. As inputs and outputs have meaning and properties in the environment, they also need to be specified and possibly relevant aspects of the environments. All the knowledge that is required here, needs to be available in the used formal specification formalism. That formal descriptions which are prerequisite for the actual specification are a necessary background theory and providing that in sufficient detail

in order to allow to specify and possibly verify the requirements is a challenge. Further challenges are to come up with lemmata which are necessary to automatically prove domain properties, or annotations to programs, such as, e.g., necessary invariants.

Support offered by LLMs. The examples presented in Section 2.3 mainly address the annotation of programs and the generation of invariants. The interaction between the LLM generating the annotations and the verifier checking and possibly rejecting them is a sensible approach to continue on. What is actually entirely missing in the literature are approaches how a background theory could be obtained.

Requirements for formal methods. Formal verification tools can be used to analyse the formal specifications generated by the large language model and verify that the system meets the safety requirements. Digital twins used as digital models can be used to simulate the behaviour of the system and provide a visual representation of the system's dynamics, allowing engineers to analyse and understand the system's behaviour and identify potential safety issues. This is in line with most usages of digital twins identified in the work presented in Section 3. Large language models can be used to analyse the output of the formal verification process and provide a natural language summary of the results, making it easier for engineers to understand the results and identify areas for improvement.

Training data for pre-training LLMs. A possible avenue to explore how to obtain high-quality domain theories is the infusion of domain knowledge into LLMs. Currently, the background or domain knowledge can be included via fine tuning the models, integration of knowledge graphs or adding memory, e.g., in the form of key value pairs (Jha et al. [16]). A good survey on knowledge graph embedding is provided by Wang et al. [31]. All this happens in the context of neurosymbolic AI, where, e.g., Glauer et al. [12] provide an interesting approach to fuse LLMs and ontologies (there for the domain to detect poisonous chemical substances, but the method is more general). Such activities of including domain knowledge in LLMs could then be combined with the activities on theory exploration that was coined by Buchberger [4] (see Johansson [18] for more recent work).

4.3 Software / Program Synthesis

Challenge using formal methods. Compared to the programming CoPilots that are currently proliferating the challenge for software/program synthesis in formal methods is that not only the software is syntactically and statically correct, but also fulfils the formal properties.

Support offered by LLMs. The LLM support provided in Section 2.4 are based on the means developed in formal software synthesis consisting of enumerating programs. LLMs are included to improve that process, but not to fundamentally

change the process. This could be reconsidered by, e.g., more tightly combine specifications and annotated programs based on examples. A conversational approach between the programmer, the LLM generating annotated software and a verifier (as done in the examples from Section 2.3) checking or rejecting annotated software could be a sensible approach. Blocklove et al. [2] present such work in the context of hardware design exploration, for instance.

Requirements for formal methods. For a formal method verification tool to be included in the interaction with the LLMs the information returned in case of rejecting a solution (i.e., a program) gets more crucial. Indeed, that information shall be used by the LLM in order to gear it towards a better / correct solution. To date no research is known how in a systematic and structured manner negative feedback of a verifier shall be provided depending on the task at hand in order to assist the user (or then the LLM).

Training data for pre-training LLMs. Published annotated programs / software could serve as a basis to pre-train LLMs. Similarly to the problem to formalize natural language requirements from Section 4.1, the problem may be that such data is not publicly available, e.g., because it may contain sensible information. Again, this may require thinking in the direction to protect the data by using approaches like federated learning with dynamic federations of participating institutes and industry.

An enhancement in the direction of providing more ascertained relationships between software and formalizations is to not only provide, e.g., pre- and postconditions, but fully annotated program code, where all intermediate points in the program are annotated with properties holding at this point. This information is typically computed by verification condition generators and could be included. It would be something like *properties carrying code* in the spirit of proof carrying code (Necula and Lee [23]) and, aside from providing more knowledge for LLMs on the relationship between code and formal properties, maybe also be used to allow for LLMs to suggest correct-by-construction programs.

4.4 Continuous Monitoring

Challenge using formal methods. As presented in Section 3, digital twins are not taken advantage of during operation, e.g., for purposes of self-adaptation. Though there are no reasons indicated by Löcklin et al. [21], one issue could be that formal method representations are different from the data representation of the actual system. Hence to use formal methods techniques, e.g., for analysis or synthesis, would need a mapping of the real data into formal representations.

Support offered by LLMs. LLMs support could be envisioned in different aspects for systems in operation. First, LLMs could be used to monitor the behaviour of the system in real-time and detect potential safety issues, that may for instance result from deficient components. LLMs can be used to analyse the output of the monitoring process and provide a natural language summary of

the results, making it easier for engineers to understand the results and identify areas for improvement. Furthermore, in case operational systems need to be adapted, e.g., because additional requirements arise, components need to be exchanged, or some components are disfunctional, there is a need to come up with a plan how to go from the current system towards the new system, while ensuring different system properties during the transformation process.[5] Performing such search in combination with formal methods verifying the properties can be a long process, for which there may not be enough time, because the reactions need to happen quickly, i.e., to restore again full operation of the overall systems. In this case LLMs, pre-trained on appropriate data, could serve to suggest the adaptions and transformations (similar to the approaches presented by Jha et al. [16] (see Section 2.4): these could be based on a digital model of the real system (or digital shadows and twins taken offline that are then usable as digital models) could be used to simulate the effect of adaptations proposed by the LLM in interaction with a verifier checking proposed solutions, until viable solutions are found and can be proposed to the users.

Training data for pre-training LLMs. Digital models (or, again, digital shadows and twins taken offline that are then usable as digital models) of the real system can be used to simulate the behaviour of the system and provide a visual representation of the system's dynamics, allowing engineers to analyse and understand the system's behaviour and identify potential safety issues. Moreover they could serve to systematically generate adaptations of a system and assess safety or security properties. Those adaptations could serve as training data for LLMs to suggest safe variants of a given systems or steps to take to adjust a given system at risk towards obtaining again a safe system.

In summary, large language models and digital twins/shadows that consist of an executable digital model could be used cooperatively to automate creative tasks in providing safety properties and formal verification of real-world systems. They could be used to generate formal specifications, verify the safety of the system, develop safety cases, generate test cases, and monitor the behaviour of the system in real-time.

5 Discussion

Motivated by the increasing adoption of large language models with good generative capabilities for many tasks, we have reconsidered the reasons why the use of formal methods is not so common practice. Along the life cycle of software development and deployment we reviewed the literature how LLMs are used in

[5] Systems that have been formally verified may be disfunctional in operation. This can be due to the abstraction necessarily made to formalize the actual system in operation. Also latencies in communication or deficiencies in hardware may be reasons. Overall, there are despite verification possibilities for failures or attacks not visible during verification which could be summarized as *simulation to reality gap* as known, e.g., from robotics systems.

different tasks or phases. That literature turns out to be very recent, some only published in pre-prints at time of writing this paper, which indicates a growing interest of the research community in that topic in general. A general scheme in the use of LLMs with formal methods is conversational, i.e., where a conversation is created between the LLMs that provide suggestions for the specific task and formal method verifiers assessing the suggestions and, in case of rejection, provide feedback to the LLMs in a form that allows them to adjust their suggestions. However, nearly all reported activities though promising are either very narrow in the target application domain or still at the beginning of the whole endeavour.

Following that, we have identified for each specific tasks in the life cycle what makes the adoption and use of formal method techniques difficult and cumbersome. For the different tasks we discussed the role LLMs could play, how the interaction with the formal methods tools could be, and how data could be generated. Regarding the interaction between a formal method technique and LLMs, an interesting research avenue to explore systematically how, for the task at hand, formal methods should provide their feedback in case they reject suggestions of the LLM. Regarding the generation of data to pre-train LLMs, we have proposed new ideas. Especially to support requirements elicitation, engineering of formal specifications and continuous monitoring and adaptation at runtime, digital models (possibly obtained by taking offline real digital twins or digital shadows) in combination with formal method checking tools could beneficially be used to create training data to pre-train LLMs offline. Especially for continuous monitoring, where currently digital twins are not used (but maybe digital models or shadows are available) having LLMs available could foster the application of formal methods in such time critical situations.

References

1. Z. Azerbayev, H. Schoelkopf, K. Paster, M. D. Santos, S. McAleer, A. Q. Jiang, J. Deng, S. Biderman, and S. Welleck. Llemma: An Open Language Model For Mathematics. pages 1–28, oct 2023. URL http://arxiv.org/abs/2310.10631.
2. J. Blocklove, S. Garg, R. Karri, and H. Pearce. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6. IEEE, sep 2023. ISBN 979-8-3503-0955-3. https://doi.org/10.1109/MLCAD58807.2023.10299874.
3. T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 2020-Decem, may 2020. ISSN 10495258. URL http://arxiv.org/abs/2005.14165.

4. B. Buchberger. Theory exploration with theorema. analele universitatii din timisoara, ser. matematica. *Informatica*, 38:4–6.
5. D. R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6617 LNCS, pages 472–479. 2011. ISBN 9783642203978. https://doi.org/10.1007/978-3-642-20398-5_35.
6. L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean Theorem Prover (System Description). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9195, pages 378–388. 2015. ISBN 9783319214016. https://doi.org/10.1007/978-3-319-21401-6_26.
7. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. oct 2018. URL http://arxiv.org/abs/1810.04805.
8. E. First, M. Rabe, T. Ringer, and Y. Brun. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241, New York, NY, USA, nov 2023. ACM. ISBN 9798400703270. https://doi.org/10.1145/3611643.3616243.
9. N. E. Fuchs, K. Kaljurand, and G. Schneider. Attempto controlled english meets the challenges of knowledge representation, reasoning, interoperability and user interfaces. *FLAIRS 2006 - Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, 2006: 664–669, 2006. URL https://aaai.org/papers/flairs-2006-131/.
10. M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 888–891, New York, NY, USA, sep 2018. ACM. ISBN 9781450359375. https://doi.org/10.1145/3238147.3240481.
11. M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. ISBN 978-1-107-03727-4. URL http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/automated-planning-and-acting?format=HB.
12. M. Glauer, F. Neuhaus, T. Mossakowski, and J. Hastings. Ontology Pre-training for Poison Prediction. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 14236 LNAI:31–45, 2023. ISSN 16113349. https://doi.org/10.1007/978-3-031-42608-7_4.
13. M. W. Grieves. Virtually Intelligent Product Systems: Digital and Physical Twins. In *Complex Systems Engineering: Theory and Practice*, number July, pages 175–200. American Institute of Aeronautics and Astronautics, Inc., Reston, VA, jan 2019. ISBN 9781624105654. https://doi.org/10.2514/5.9781624105654.0175.0200.

14. M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate Automizer with SMTInterpol. pages 641–643. 2013. https://doi.org/10.1007/978-3-642-36742-7_53.
15. IEC. *IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems, Parts 1 to 7*. International Electrotechnical Commission, 2.0 edition, 2010.
16. S. Jha, S. K. Jha, P. Lincoln, N. D. Bastian, A. Velasquez, and S. Neema. Dehallucinating Large Language Models Using Formal Methods Guided Iterative Prompting. In *2023 IEEE International Conference on Assured Autonomy (ICAA)*, pages 149–152. IEEE, jun 2023. ISBN 979-8-3503-2601-7. https://doi.org/10.1109/ICAA58325.2023.00029.
17. A. Q. Jiang, W. Li, J. M. Han, and Y. Wu. LISA: Language models of ISAbelle proofs. *6th Conference on Artificial Intelligence and Theorem Proving*, pages 378–382, 2021.
18. M. Johansson. Automated theory exploration for interactive theorem proving: An introduction to the hipster system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10499 LNCS:1–11, 2017. ISSN 16113349. https://doi.org/10.1007/978-3-319-66107-0_1.
19. W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn. Digital Twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, 51(11):1016–1022, 2018. ISSN 24058963. https://doi.org/10.1016/j.ifacol.2018.08.474.
20. Y. Li, J. Parsert, and E. Polgreen. Guiding Enumerative Program Synthesis with Large Language Models. 2024. URL http://arxiv.org/abs/2403.03997.
21. A. Löcklin, M. Müller, T. Jung, N. Jazdi, D. White, and M. Weyrich. Digital Twin for Verification and Validation of Industrial Automation Systems - A Survey. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2020-September:851–858, 2020. ISSN 19460759. https://doi.org/10.1109/ETFA46521.2020.9212051.
22. L. Ma, S. Liu, Y. Li, X. Xie, and L. Bu. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. jan 2024. URL https://arxiv.org/abs/2401.08807v1 http://arxiv.org/abs/2401.08807.
23. G. C. Necula and P. Lee. Proof-Carrying Code. Technical Report November, School of Compiter Science, Carnegy Mellon University, Pittsburgh, PA 15213, 1996. URL http://www.eecs.berkeley.edu/ necula/Papers/tr96-165.ps.gz.
24. E. Negri, L. Fumagalli, and M. Macchi. A Review of the Roles of Digital Twin in CPS-based Production Systems. *Procedia Manufacturing*, 11(June):939–948, 2017. ISSN 23519789. https://doi.org/10.1016/j.promfg.2017.07.198.
25. C. Paulin-Mohring. Extracting ω's programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*, pages 89–104, New York, New York, USA, 1989. ACM Press. ISBN 0897912942. https://doi.org/10.1145/75277.75285.

26. K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin. Can Large Language Models Reason about Program Invariants? In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR, 2023. URL https://proceedings.mlr.press/v202/pei23a.html.
27. K. Rajkovic and E. Enoiu. NALABS: Detecting Bad Smells in Natural Language Requirements and Test Specifications. pages 8–10, feb 2022. URL http://arxiv.org/abs/2202.05641.
28. P. Song, K. Yang, and A. Anandkumar. Towards Large Language Models as Copilots for Theorem Proving in Lean. (NeurIPS):1–9, apr 2024. URL http://arxiv.org/abs/2404.12534.
29. C. Szegedy. A Promising Path Towards Autoformalization and General Artificial Intelligence. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12236 LNAI, pages 3–20. Springer International Publishing, 2020. ISBN 9783030535179. https://doi.org/10.1007/978-3-030-53518-6_1.
30. H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models. jul 2023. URL http://arxiv.org/abs/2307.09288.
31. Q. Wang, Z. Mao, B. Wang, and L. Guo. Knowledge Graph Embedding: A Survey of Approaches and Applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, dec 2017. ISSN 1041-4347. https://doi.org/10.1109/TKDE.2017.2754499.
32. Q. Wang, C. Kaliszyk, and J. Urban. First Experiments with Neural Translation of Informal to Formal Mathematics. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11006 LNAI, pages 255–270. Springer International Publishing, 2018. ISBN 9783319968117. https://doi.org/10.1007/978-3-319-96812-4_22.
33. M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle Framework. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5170 LNCS, pages 33–38. 2008. ISBN 3540710655. https://doi.org/10.1007/978-3-540-71067-7_7.

34. H. Wu, C. Barrett, and N. Narodytska. Lemur: Integrating Large Language Models in Automated Program Verification. pages 1–19, oct 2023. https://doi.org/https://doi.org/10.48550/arXiv.2310.04870.
35. Y. Wu, A. Q. Jiang, W. Li, M. N. Rabe, C. Staats, M. Jamnik, and C. Szegedy. Autoformalization with Large Language Models. *Advances in Neural Information Processing Systems*, 35(NeurIPS):1–16, may 2022. ISSN 10495258. URL http://arxiv.org/abs/2205.12615.