Information and Software Technology 181 (2025) 107697

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Formal requirements engineering and large language models: A two-way roadmap

Alessio Ferrari [a,b,*], Paola Spoletini [c]

[a] Consiglio Nazionale delle Ricerche (CNR), Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" (ISTI), Pisa, Italy
[b] University College Dublin (UCD), School of Computer Science, Dublin, Ireland
[c] Kennesaw State University, GA, USA

ARTICLE INFO

Keywords:
Requirements engineering
Formal methods
Large language models
LLMs
Natural language processing
NLP
NLP4RE
Prompt engineering
Prompt requirements engineering

ABSTRACT

Context: Large Language Models (LLMs) have made remarkable advancements in emulating human linguistic capabilities, showing potential also in executing various requirements engineering (RE) tasks. However, despite their generally good performance, the adoption of LLM-generated solutions and artefacts prompts concerns about their correctness, fairness, and trustworthiness.
Objective: This paper aims to address the concerns associated with the use of LLMs in RE activities. Specifically, it seeks to develop a roadmap that leverages formal methods (FMs) to provide guarantees of correctness, fairness, and trustworthiness when LLMs are utilised in RE. Symmetrically, it aims to explore how LLMs can be employed to make FMs more accessible.
Methods: We use two sets of examples to show the current limits of FMs when used in software development and of LLMs when used for RE tasks. The highlighted limitations are addressed by proposing two roadmaps grounded in the current literature and technologies.
Results: The proposed examples show the potential and limits of FMs in supporting software development and of LLMs when used for RE tasks. The initial investigation into how these limitations can be overcome has been concretised in two detailed roadmaps for the RE and, more largely, the software engineering community.
Conclusion: The proposed roadmaps offer a promising approach to address the concerns of correctness, fairness, and trustworthiness associated with the use of LLMs in RE tasks through the use of FMs and to enhance the accessibility of FMs by utilising LLMs.

1. Introduction

Large Language Models (LLMs) [1] have shown remarkable skill in mimicking human linguistic abilities. These models are capable of processing and generating human-like text based on learned patterns, which has led to their widespread application in numerous fields, including software engineering (SE). In SE, LLMs contribute to various activities such as code synthesis, documentation creation, and ideation, thereby improving the efficiency of processes that traditionally require extensive human effort [2,3]. Despite slower initial adoption in requirements engineering (RE) compared to other SE fields, recent literature demonstrates significant potential for LLMs to enhance efficiency and tackle traditional challenges [3]. Recent growth in LLM research within RE includes summarisation of requirements [4], user story generation [5], traceability support through advanced prompting strategies [6] and model generation [7–9]. While these approaches deliver promising results, questions arise about the reliability, correctness, and interpretability of the results generated by these models. To address these concerns, we propose using formal requirements and formal verification techniques to provide guarantees on the LLM-assisted activities.

Formal requirements provide a structured and clear specification of a system's expected behaviour and constraints [10]. They precisely outline system functionalities and qualities. Unlike informal requirements, which can be ambiguous, formal requirements use mathematical or logical language to minimise the risk of misinterpretation. The adoption of formal requirements enables the use of formal methods (FMs), which are mathematical techniques for analysing and verifying systems. These methods, such as model checking [11], abstract interpretation [12], and deductive verification [13], aim to increase the rigour and correctness of the software development process. However, due to the specialised knowledge required to apply formal requirements and their

* Corresponding author.
E-mail addresses: alessio.ferrari@isti.cnr.it (A. Ferrari), pspoleti@kennesaw.edu (P. Spoletini).

https://doi.org/10.1016/j.infsof.2025.107697
Received 10 July 2024; Received in revised form 6 February 2025; Accepted 7 February 2025
Available online 18 February 2025

associated verification techniques, their use is predominantly confined to the development of critical and complex systems.

Given the potential central role of formal approaches in software development, especially for complex and critical systems, our vision also includes steps to make formal RE more accessible. To this end, we propose to use LLMs to support the formalisation of requirements, explain formal requirements, and help interpret the results of formal tools.

In this paper, we extend our initial vision presented in [14] by providing the following contributions:

- An overview of LLMs and their applications in RE, along with a discussion on formal RE[1] (Section 2);
- A comprehensive roadmap to use LLMs to make them more usable in practice (Section 4) supported by a simple example of the use of formal RE in software development (Section 3);
- A detailed roadmap to use formal RE to overcome the limitations of LLMs in supporting RE activities (Section 6), motivated by a set of examples illustrating the use of LLMs in the software engineering process (Section 5); and
- Considerations on the practical implementation of the roadmaps, with associated risks and limitations (Section 7).

It is worth remarking that this is a vision paper, which does not aim to offer sound empirical evidence but rather to indicate possible avenues of research. Nevertheless, we have made an effort to make the vision practical by pointing to seminal works that can shape future research and by including practical considerations as well as potential barriers to realising the vision. We also want to emphasise that, despite our efforts to touch upon multiple relevant points, the discussed roadmaps should not be considered exhaustive. They reflect the opinions and experience of the authors, who have a background in RE, FMs, and natural language processing (NLP). Moreover, it is designed to encourage further reflection, with the potential to either expand upon its content or challenge the plausibility and relevance of certain research directions.

2. Background

In this section, we provide an overview of the field of LLMs, also including applications to the RE field, and some background on formal RE.

2.1. Large language models and requirements engineering

A language model (LM) is a computational model designed to represent human language by capturing the statistical structure and semantic relationships within a corpus of textual data. The construction of an LM encompasses various techniques and methodologies that have evolved over time. In the following, we present a brief history of LMs, followed by a discussion of studies using LMs in RE.

Traditional language models. The earliest LMs, such as bag-of-words (BoW) and term frequency-inverse document frequency (tf-idf), focus on representing textual documents based on the frequency of individual words [15]. Given a document,[2] BoW disregards the order of words in the document and treats it as an unordered collection of words. This leads to a vector representation for documents, in which each component of the vector is associated with a word, and the content of the component is the frequency of occurrence of the word in the document. Instead, tf-idf emphasises the *importance* of words in a document relative to their occurrence across a corpus of documents. The frequency of a word is weighted based on its frequency in the corpus, with the assumption that a word that occurs more frequently is less informative (i.e., less characterising for the document) than a word that occurs less frequently. Given a document, this is again represented as a vector where each word is a component, but the content of the component is the word frequency divided by the frequency of the word in the corpus. These traditional LMs are typically used to compare their similarity for the purpose of information retrieval or clustering. The similarity is typically computed in the vector space induced by the vectorial representation of documents, typically by means of the *cosine similarity* [15]. The closer the cosine between the vectors is to one, the more similar are the compared documents.

Word embeddings. Traditional LMs suffer from the lack of representation of the *meaning* of the words that they were representing. In the vector spaces induced by these representations, sentences using different words but having similar meanings would not be considered similar to each other, as their vectors would be completely different. To solve this problem, word embeddings were introduced in 2013 by Mikolov et al. [16], following the distributional semantics idea that the meaning of a word is determined "by the company it keeps" [17], which implies that words derive their meaning from the context in which they are used rather than from abstract definitions. According to this vision, each word is represented as a dense vector in a continuous vector space. In this vector space, words that have a strong association, because they typically occur in similar textual contexts (i.e., together with the same words), are represented through vectors that are closer to each others. These vectors are built through a shallow feedforward neural network trained on a large textual corpus, with a task that enables the network to learn the typical contexts where words occur. To this end, the learning of word embeddings LM is based on a training task where the objective is to predict surrounding words given a target word or vice versa. `word2vec` [16] is one of the first algorithms for word embeddings, and was followed by GloVe, FasText, Universal Sentence Encoder (USE), and others—see Johnson [18] for a summary. Once words are represented with these types of LMs, their representation can be used to perform other downstream tasks, e.g., sentence classification, language translation, etc., typically using the vectors as input to train task-specific machine learning algorithms.

Pre-trained transformers and BERT. In word embeddings, each word is represented as a fixed, dense vector in a continuous vector space based solely on its co-occurrence statistics within a large corpus of text. These embeddings capture semantic similarities between words based on their distributional patterns in the training data. However, word embeddings lack contextual information about how a word's meaning might change depending on its surrounding words in a specific sentence or context. As a result, words with multiple meanings or usages may have a single representation, leading to ambiguity in capturing the nuanced meaning of words in different contexts. To solve this issue, the Bidirectional Encoder Representations from Transformers (BERT) architecture was introduced [19]. BERT employs a transformer architecture, which is a deep neural network specifically designed for sequential data processing. Using this type of architecture in NLP tasks typically consists of two phases: *pre-training*, i.e., learning an LM from a document corpus, and *fine-tuning*, i.e., further training the learned LM on a specific downstream task, such as classification or information extraction, with task-specific annotated data.

During pre-training, BERT learns contextualised representations of words from a corpus by considering the entire input sentence bidirectionally. This means that each word's representation is dynamically influenced by its surrounding words in the sentence, allowing BERT

[1] With the term formal RE, we characterised all those mathematical-based techniques that specify, model, and verify software requirements, such as techniques to formally express properties and describe software models and tools for verification.

[2] The term "document" here has the general meaning of "finite sequence of tokens representing a single conceptual entity", where tokens are words, punctuation, spaces, numbers, etc. In other terms, a document can also be a sentence, paragraph, query, Tweet, or any individual unit of text that conveys a coherent message or idea.

to capture intricate semantic relationships and contextual nuances. BERT achieves this bidirectional contextual understanding through a process called *masked language modelling*. During pre-training, BERT randomly masks some of the input words in a sentence and tasks the model with predicting the masked words based on the surrounding context. Furthermore, BERT incorporates the concept of *self-attention* mechanisms, which allows it to weigh the importance of different words in a sentence dynamically based on their contextual relevance. This attention mechanism enables BERT to focus more on informative words while disregarding irrelevant ones, leading to richer and more contextually relevant representations of text. After pre-training, BERT-based word representation learned from an input corpus can be used, e.g., to encode sentences. The encoding of a word will then vary based on its context in the input sentence—i.e., surrounding words and syntactic structure.

Pre-trained BERT models are then fine-tuned on specific downstream tasks, such as text classification, named entity recognition, and question answering, by adding task-specific layers on top of the pre-trained BERT architecture. This fine-tuning process adapts the learned word representations to the specific characteristics of the target task. Fine-tuning can be performed also on domain-specific documents, so that, instead of learning a task, the LM can improve its understanding of the language of the domain. Based on the BERT architecture, different alternatives have been proposed, such as RoBERTa (Robustly optimised BERT approach), XLNet, Electra (Efficiently Learning an Encoder that Classifies Token Replacements Accurately), and others—see Kalyan et al. [20] for a survey.

Large language models (LLMs). The transformer-based architecture of BERT is also used in today's LLMs, such as GPT, Llama, Mixtral, Gemini, and others—see Minaee et al. for a survey [21]. LLMs are typically larger in terms of the number of parameters (e.g., weights and biases in the neural network) and natural network layers compared to BERT and previous transformers. These larger models require more computational resources for training and inference. For example, while BERT in its base version has around 110 million parameters, it is estimated that GPT-4 could have around 100 trillion parameters. LLMs are designed to excel across a wide range of NLP tasks without extensive task-specific fine-tuning, which is typically required for BERT-like LMs. LLMs demonstrate impressive capabilities in downstream tasks without examples (zero-shot learning) or with few examples of the task input and output (few-shot learning). LLMs are frequently based on the question-answering paradigm, where the downstream task is expressed with a *prompt*, i.e., a natural language (NL) query or instruction, which the LLM is required to answer or perform. The introduction of this paradigm has led to the development of a large variety of prompt strategies, such as chain of thought (CoT) prompting [22], generated knowledge prompting [23], see Quiao et al. [24] for a survey. Further techniques for exploiting the capabilities of LLMs, and especially avoiding costly re-training operations, include instruction fine-tuning [25], retrieval augmented generation (RAG) [26], Low-Rank Adaptation (LoRA) [27], and distillation [28].

Instruction fine-tuning, or simply instruction tuning, aims to improve the LLM's ability to follow instructions expressed in NL and consists of fine-tuning the LLM to perform instruction-based tasks. In the more classical LM fine-tuning, the training dataset typically consists of a data item with associated annotations (e.g., a requirement and its class). With instruction tuning, the training dataset consists of a NL instruction, which specifies the task to be addressed (e.g., what is the class of the following requirement?) and the expected answer (e.g., the actual class). This has been shown to improve the capability of the LLM to follow specific instructions [29].

RAG enhances the capability of LLMs to produce accurate and context-relevant output by incorporating external information in the generation process. When tasked with a certain prompt, prompt-relevant information is first retrieved from an external knowledge source. The original prompt is enhanced with the additional information, and the LLM is queried with this prompt to produce the desired output. This enables the possibility to combine the language understanding of LLM with external, typically domain-specific knowledge.

LoRA is a technique used to adapt large pre-trained language models to specific tasks or domains without retraining all the model parameters. LoRA starts by freezing the weight matrices of the pre-trained LLM, which means these weights are not updated during the adaptation process. LoRA introduces additional matrices into the model that are smaller in size compared to the model's original weights. These matrices are trainable, meaning that they can be adjusted during the fine-tuning process based on the task-specific data. The fine-tuning does not modify the weights of the original LM, while modifies the output of the large matrices during the forward pass of the neural network.

Finally, distillation, or knowledge distillation, is a technique aimed at transferring the knowledge and capabilities from a large, complex model (often referred to as the *teacher*) to a smaller, more efficient model (known as the *student*). The goal is to enable the student model to perform as closely as possible to the teacher model while being less resource-intensive.

Another relevant issue to address is the complexity of certain tasks, which cannot be addressed with a single LLM, but require multiple LLMs or external resources. To cope with this problem, LLM agents have emerged [30]. An LLM agent is a system that not only generates human-like text but also has additional abilities to handle sophisticated, multi-step workflows, perform reasoning, remember information, and use tools or external systems, e.g., through APIs, to complete specific tasks. In a system based on LLM agents, a main LLM acts as the central controller or "brain", orchestrating a sequence of actions required to fulfil a task or user request, leveraging other LLM agents. LLM agents can decide when to use certain modules (e.g., retrieving stored knowledge or performing numerical calculations) and adjust their approach based on the task at hand. Frameworks supporting the development of LLM agents are LangChain.[3] and AutoGPT.[4]

LMs and LLMs in RE. NLP techniques in general and LMs, in particular, have been widely studied and applied in RE since its early days—the first works appeared already in the '80s [31]. Typical RE tasks that have been addressed with the support of LMs are ambiguity and smell detection [32], tracing [33], classification of requirements [34], as well as app reviews [35] and issues [36], and other tasks. Early studies using tf-idf and BoW focused on solutions for tracing and for requirements retrieval from large repositories. In these studies, these prototypical LMs were mainly exploited to measure the similarity between requirements so that trace links could be identified and requirements could be retrieved from existing repositories with the goal of requirements reuse. Word embeddings were also used, specifically for the identification of cross-domain ambiguity, i.e., terms that have different meanings in different domains, and to support requirements tracing with the aid of recurrent neural networks [33]. BERT-like transformers were first exploited for requirements classification [34,37,38], and saw applications in a multitude of tasks, including requirements retrieval [39], app review analysis [40], and assessment of requirements completeness [41].

The uptake of LLM-based techniques has been rather slow in RE compared to other SE subfields [3], although some exploratory work has been started. Let us consider some relevant contributions. Among the early works, Jain et al. [4] presents a set of experiments on the usage of different LLMs for requirements summarisation from legal texts. They first generated summaries with the aid of GPT-3 and used them to fine-tune Pegasus, GPT-2, and other open-source models that do not require the disclosure of data, as in the case of GPT-3. Then, they evaluated the performance of these models on industrial datasets, showing

[3] https://www.langchain.com/ Last visit 30 Oct., 2024.

[4] https://autogpt.net/category/autogpt/. Last visit 30 Oct., 2024.

that Pegasus outperforms the other options. Marczak et al. [5] use ChatGPT to generate user stories reflecting human values. The user stories are then used as creativity triggers in requirements elicitation sessions. The evaluation with focus groups shows that the generated user stories effectively inspire participants to identify value-relevant requirements. Rodriguez et al. [6] experiment with prompting strategies to support requirements traceability with the aid of Anthropic's Claude instant model. Their exploratory evaluation with the CM1 dataset shows that small prompt modifications can lead to significant differences in model outputs.

Some works also explore the potential of LLMs for model generation. Among them, Chen et al. [7] evaluates the potential of GPT-4 for generating goal models using the textual grammar for the Goal-oriented Requirement Language (GRL) based on free-form NL descriptions of the problem context. Their experiments on four example cases show that incremental prompts can improve the results, and domain knowledge is highly needed to assess the plausible—yet frequently incorrect—output. Chen et al. [42] evaluates GPT-3.5 and GPT-4 for generating class diagrams based on NL descriptions. Their evaluation on a ground truth from an undergraduate course shows that the LLMs struggle to generate relationships between classes. Furthermore, they show that while few-shot approaches (i.e., adding examples to the prompts) can improve performance, chain-of-thought prompting (i.e., adding reasoning steps) leads to a performance decrease. Still on class diagram generation, Camára et al. [8] perform an exploratory study using ChatGPT. They exercise the model with different sample problems and provide some lessons learned. The authors conclude that iterative prompting is needed to produce models of sufficient quality, and adherence to the standard in terms of construct semantics tends to be poor. Finally, the study of Ferrari et al. [9] explores the capability of ChatGPT to generate UML sequence diagrams from requirements. The authors observed that, although the models generally conform to the standard and exhibit a reasonable level of understandability, their correctness with respect to the specified requirements often presents challenges. This issue is particularly pronounced in the presence of requirements smells, such as ambiguity and inconsistency.

Besides these initial efforts, recent months have seen substantial growth in the literature in the area of LLMs for RE, as is evident from the accepted papers at RE@Next! 2024, which includes seven papers about LLMs out of 19 contributions, and at RE 2024, which includes eight papers out of 14 contributions.[5] These papers cover a variety of RE tasks through the use of LLMs, from an approach that uses GPT for automating text completion of requirements within the context of the specification [43] to the use of in-context learning to support the evaluation of satisfaction arguments, which describe how a requirement is satisfied by a system specification [44]. Concerning the task of requirements elicitation, Gorer et al. propose an approach that uses LLMs to generate interview scripts [45]. The use of LLMs is also proposed to elicit and specify SRS documents [46] and check the quality of requirements [47]. Additional exploratory studies investigate the use of LLMs to generate code from requirements [48] or to improve the representation of normative requirements [49].

2.2. Formal methods and formal RE

When formal techniques are applied, the system requirements are represented through a formal specification that defines "what" the system should do and the constraints it must operate under (e.g., [50]). Then, either this specification is directly analysed, or the system is modelled through a state-based formalism, which describes "how" the system operates (e.g., [51]) and then analysed against properties extracted from the specification and often represented in mathematical logic (e.g., [52]). In the following, we will present some of the most developed and used formal techniques to specify requirements, create models, and analyse them.

[5] https://conf.researchr.org/home/re-2024. Last Visit 29 Oct. 2024.

Formal representation of specifications. FMs for requirements specification were first used in the 1970s to cope with the need for more rigorous and precise software development methods, given the increasing complexity and critical use of software systems in fields such as aerospace, medical, and military domains. In addition, high-profile software failures (e.g., the partial meltdown of the Three Mile Island nuclear power plant in Pennsylvania [53]) highlighted the limitations of conventional development techniques and the need for more precise and verifiable approaches to software development.

An example of formal languages for specifications is the *Z Notation* [54,55], which uses set theory and first-order predicate logic to specify system properties and behaviours. The core construct in the Z notation is the schema, which groups related state variables and their invariants together. Schemas can be used to describe both the state of a system and the operations that can be performed on that state. They can also be composed to support modularity, allowing complex systems to be described in a structured and hierarchical manner.

Analogously to the Z notation, the *Vienna Development Model* (VDM) [56] provides a model-based approach that simplifies the design and verification of complex systems. VDM uses formal specification languages based on mathematical notation to describe the system's functionality, structure, and constraints. The most commonly used specification languages in VDM are VDM Specification Language [57] and VDM++ [58], an object-oriented extension of VDM-SL. VDM crates mathematical models as blueprints for the system, capturing its essential properties and behaviours. VDM emphasises the use of formal proofs to verify that the system's implementation satisfies its specifications. Proof obligations are generated to demonstrate that the system is correct with the support of tools like VDMTools [59] and Overture [60].

Differently from the Z notation and VDM, which primarily focus on providing a formal specification language, the *B-Method* [61] is a comprehensive formal method covering specification, refinement, and implementation with a strong emphasis on proofs and tool support. The core concepts in the B-Method are abstract machines, which represent components or modules of a system, define the state of the system through variables, and specify operations that can change this state. The operations are specified through preconditions that must be true before an operation can be executed and postconditions that describe the state after the operation has been executed. The B-Method allows transforming an abstract specification into a more concrete one, step by step, until an executable implementation is achieved through subsequent refinements.

In addition to these approaches, over the last three decades, a variety of languages have been introduced. *Process algebras* such as CCS (Calculus of Communicating Systems) [62] and CSP (Communicating Sequential Processes) [63] are formal languages used to describe and analyse the behaviour of concurrent systems. The *Specification and Description Language* (SDL) [64] is another formal language specifically designed for describing complex, event-driven, real-time systems, commonly used in telecommunications and other industries. The *Common Algebraic Specification Language* (CASL) [65] and *Language of Temporal Ordering Specification* (LOTOS) [66], including its extended version E-LOTOS, are algebraic and process-based languages used for specifying and verifying concurrent and distributed systems. *Temporal Logic of Actions* (TLA+) [67] and *Alloy* [68] are used for specifying and modelling systems, with Alloy focusing on relational logic and TLA+ on temporal logic. Finally, some specification languages are inspired by programming languages or natural languages. For example, *FizzBee* [69] is a Python-inspired design specification language. *Attempto Controlled English* (ACE) [70] is a controlled natural language that allows specifications to be written in a subset of English, aiming to reduce the complexity of formal methods for non-expert users.

Representing the properties of the system. The properties that a specification has to satisfy (e.g., safety or liveness) are usually represented using some (temporal) extension of propositional or predicate logic [71]. *Linear Temporal Logic* (LTL) [72] is an extension of propositional logic, widely used for specifying properties over linear sequences of states. LTL is particularly effective for expressing safety and liveness properties, which describe what must or must not happen during system execution. The syntax of LTL includes temporal operators such as X (next), F or ⟨⟩ (eventually), G or [] (globally), and U (until). *Computation Tree Logic* (CTL) [73] is another temporal logic used to specify properties over branching structures of states, which allows for the expression of properties concerning possible future states. It combines path quantifiers A (for all paths) and E (there exists a path) with the same temporal operators used in LTL. In CTL, path quantifiers cannot be used independently from temporal operators, and this makes LTL's and CTL's expressiveness incomparable. Instead, *CTL** [74] combines both LTL and CTL, allowing more complex specifications by enabling both path quantifiers and temporal operators to be mixed freely. *μ-calculus* [75] is another powerful formalism that extends temporal logics like LTL and CTL by introducing fixed-point operators. This makes it one of the most expressive logics for specifying properties of transition systems, enabling the expression of complex properties like those involving recursion.

Higher-order logic (HOL) [76] extends predicate logic by allowing quantification over predicates and functions, providing a more expressive framework for specifying and verifying complex properties. Additional logical formalisms used to specify properties include *Modal Logic* [77], *Metric* (MTL) and *Real-Time Temporal Logic* (RTTL) [78], and *Probabilistic Temporal Logics* (e.g., Probabilistic Computation Tree Logic [79]). Modal Logic includes modalities such as necessity and possibility to express properties about different modes of truth, such as "necessarily true" or "possibly true". This logic is useful in systems where different states or conditions can coexist, and properties need to be expressed concerning these modes. MTL and RTTL extend LTL with timing constraints, allowing the specification of properties in real-time systems. Probabilistic Temporal Logics are used to specify properties of probabilistic systems, incorporating probabilities into the logical framework.

Formal models. *Labelled Transition Systems* (LTSs) [79] are widely used to model the behaviour of concurrent systems. An LTS consists of states and transitions between these states, where each transition is labelled with an action. Each state represents a configuration of the system, and the transition shows how to move from one configuration to another. Analogously, *Finite State Machines* (FSMs) [73] are used to model systems with a finite number of states and transitions between those states. The main difference between the two models is that FSMs use input symbols to determine state transitions, while LTSs use labelled actions that can represent a broader range of events or interactions. In addition, LTSs are well-suited for modelling concurrent systems and their interactions, whereas FSMs are more straightforward and often used for sequential logic. A particularly used type of FSMs is *Büchi Automata* [51,80], used for modelling and verifying properties of infinite sequences. These automata, as FSMs, have a finite number of states, but thanks to their acceptance condition, they can describe infinite behaviours.

FSMs can also be extended to include time and probability. *Timed Automata* (TAs) [78] extend FSMs with clocks, allowing the modelling of real-time systems. The clock variables are used to measure the passage of time and to set guards on the transitions. TAs enable the specification of timing requirements such as deadlines, periodicity, and delays. *Probabilistic* and *Stochastic State Machines* extend the classical FSMs by incorporating probabilistic and stochastic elements. These extensions enable modelling systems that exhibit random or probabilistic behaviour, which is crucial when systems operate under uncertainty. Probabilistic State Machines [81] introduce probabilities into the state transition mechanism by regulating transitions between states using predefined probabilities. Differently, Stochastic State Machines [82] incorporate timing aspects into the probabilistic state transitions, often using stochastic processes like Markov chains.

The need to describe the hierarchical structure of software has led to the introduction of different hierarchical models, such as *Statecharts* [83]. Statecharts extend FSMs by incorporating concepts of hierarchy, concurrency, and communication. They allow the definition of hierarchical states that are refined by substates to capture different levels of abstraction. In addition, they model concurrency through parallel states. Other hierarchical models include *Hierarchical State Machines* [84], *MODELICA State Machines* [85], and *Ptolemy II State Machines* [86].

A commonly used tool to model a system is *Petri Nets* (PNs) [87], a graphical and mathematical modelling tool used to describe the flow of information and control in systems, particularly in concurrent and distributed systems. PNs used places to represent conditions or states of the system and transitions to represent events causing changes in conditions. To show that a condition is met (or the system is in a state), they use tokens that allow transitions to be executed. Analogously to FSMs, PNs have been extended with different attributes to make them more suitable to represent complex systems. For example, *Colored PNs* [88] introduce colours to tokens, enabling the modelling of systems with different types of resources; *Timed PNs* [89] incorporate timing information into PNs, allowing transitions to have firing delays; *Stochastic PNs* [90] extend PNs with the probabilistic firing of transitions, incorporating uncertainty into the model; and *Hierarchical PNs* [91] introduce hierarchy into PNs by allowing places or transitions to contain subnets.

Formal methods for analysing models. There exists a variety of methods to analyse and verify the formalisms presented in this section or directly code. Given the expressive power of such formalisms and the complexity of the verification problem, each technique has different advantages and limitations as they need to compromise between the expressivity of the input languages and automation.

Abstract Interpretation [92] uses the idea that the verification of the considered system can be done at a high level of abstraction, ignoring irrelevant details. Techniques such as interval analysis and polyhedral analysis are used to approximate the behaviour of the system. Astrèe [93] and Infer [94] are examples of tools that leverage abstract interpretation to detect potential runtime errors, such as buffer overflows and division by zero, in critical software systems.

Semantic Static Analysis [95] offers a complete and automatic analysis of a program's source code without running it by using data flow analysis, control flow analysis, and type checking. This approach is supported by tools such as Coverity [96], PVS-Studio [97], and FindBugs [98] that help detect software defects and vulnerabilities and ensure compliance with coding standards by analysing source code for potential issues.

Model Checking [73] verifies specifications on different types of models of the system by complete exploration. The most commonly used approaches are state space exploration and symbolic model checking. Spin [99] is an example of an automata-based model checker that takes as input a system modelled as a Büchi Automaton (described in PROMELA, Spin input language) and a property expressed in LTL. NuSMV [100] (and its evolution nuXmv) is a symbolic model checker for the formal verification of finite-state systems, expressed in LTS, against properties specified in CTL. UPPAAL [101], instead, is used for modelling, simulation, and verification of real-time systems.

Proof Assistants are interactive tools that allow users to write proofs of theorems and are used both in mathematics and computer science. Techniques include inductive proof, coinductive proof, and automated theorem proving. Tools like Coq [102], Isabelle/HOL [103], and Agda [104] assist in developing formal proofs by providing a framework for constructing and checking proofs and have been used in verifying software correctness, cryptographic protocols, and mathematical theorems.

Deductive Verification [13] involves providing a program and a formal specification, which is then propagated through the source code of the program, using the weakest preconditions and symbolic execution. Techniques include Hoare logic, weakest precondition calculus, and predicate transformers. Examples of tools that support deductive verification are Frama-C [105], a platform dedicated to the analysis of the source code of software written in C, Dafny [106], a verification-aware programming language that allows specifications to be written in line with the code, and KeYKeY [107], a formal verification tool for Java programs.

Design by Refinement [108] is an approach based on successive refinements of a sequence of models, starting with an abstract model and ending with a concrete implementation. Techniques include step-wise refinement, data refinement, and operation refinement. Tools like Event-B [109] and Rodin, support this process by ensuring that each refinement step maintains the correctness of the system with respect to its specification. Event-B is a formal method for system-level modelling and analysis. Rodin is an open toolset for modelling and reasoning in Event-B.

3. Formal software development: an example

This section provides an overview of a possible software development process that includes formal models and tools. We start with the definition of requirements, the definition of a formal model, and the formalisation of the properties the model has to verify; then, we verify the model against the requirements through model checking and implement it. This is just a possible way of including formal RE in the development process. A different approach could actually start from the development of the code and then extract a formal model to be verified. In addition, one could use additional means, e.g., abstract interpretation/static analysis, that can be directly applied to the verification of certain properties that are specific to the code, such as, e.g., buffer overflow, or possible NULL pointers—in case languages such as C are used.

Let us consider the following requirements for a simple sender-receiver handshaking protocol that we use to guide the definition of a model and associated code:

1. **Message Integrity:** Messages sent over the protocol must arrive intact and unaltered at the receiver's end. Any tampering or corruption of the message during transmission should be detected and rejected.
2. **Order Preservation:** Messages should be delivered to the receiver in the same order in which they were sent by the sender. Ensuring that messages are received sequentially maintains the integrity of the communication flow.
3. **Flow Control:** The protocol should include mechanisms for flow control to prevent the sender from overwhelming the receiver with a large volume of messages. This could involve acknowledgements from the receiver to indicate its readiness to receive more messages.

From the requirements, we can define a formal model, e.g., in the PROMELA language [110,111], the input language of Spin, as outlined in the following listing. To simplify our example, we build the model assuming that the message content cannot be altered and messages cannot be lost, thus guaranteeing message integrity and order preservation by construction.

```
1  #define MAX_MESSAGES 10
2  chan communication = [MAX_MESSAGES] of
       {int};
3  chan control_channel = [0] of {bool};
4  int sent_message = 0;
5  int received_message = 0;
6
7  proctype Sender(chan sender_to_receiver) {
8      int counter = 0;
9      bool x;
10     do
11     :: counter < MAX_MESSAGES ->
12         sent_message=counter;
13         sender_to_receiver!sent_message;
14             counter++;
15             control_channel?eval(1);
16     ::counter == MAX_MESSAGES -> goto done;
17     od;
18     done: skip;
19 }
20
21 proctype Receiver(chan receiver_to_sender) {
22 int expected_counter = 0;
23 bool check
24 do
25     ::expected_counter < MAX_MESSAGES ->
26         receiver_to_sender?
               received_message;
27         control_channel!1;
28         expected_counter++;
29     ::expected_counter == MAX_MESSAGES ->
           break;
30 od;
31 }
32
33 init {
34     run Sender(communication);
35     run Receiver(communication);
36 }
```

Listing 1: PROMELA code for Sender and Receiver processes.

The PROMELA model consists of two processes: Sender and Receiver. The Sender process (ln 7–19) in Listing 1 sends messages to the Receiver process via the `sender_to _receiver channel`. This is done through the instruction `sender_to_receiver! sent_message`. The process pushes on the channel (with the use of the symbol `!`) the current message (`sent_message`) (cf. ln 13). Then, it waits for an acknowledgement from the Receiver (ln 15). This is achieved with the instruction `control_channel?eval(1)`, which is a blocking instruction. Indeed, the channel `control_channel` is waiting to read (symbol `?`) the value 1 (the keyword `eval` indicates that the value read on the channel needs to match what is in parenthesis, e.g., the value 1). Once it receives it, it creates a new message and sends it to the receiver. The new message will include the value of the variable `counter`, which is used to ensure the order of the messages, as this will be later controlled by the Receiver. It continues this loop until it has sent the maximum number of planned messages (ln 16). The Receiver process (ln 21–31) receives messages from the Sender process via the `receiver_to_sender channel` through the instruction in ln 26. Note that, in this case, the process reads on the channel (with the symbol `?`) a value and saves it in the variable `received_message`. Then, as it has read the message sent by the Sender, it sends an acknowledgement (the value 1 in ln 27) to the Sender. Communication between the Sender and Receiver processes is coordinated through the `control_channel`.

The initial NL requirements can also be translated into LTL formulae, which can be verified by the Spin model checker on the model in the form of assertions or never claims. Both assertions and never claims are used to verify properties, but they differ in scope and purpose. Assertions are ideal for verifying invariants, such as ensuring that a variable stays within a certain range. In contrast, never claims define undesirable global behaviours that the system should never exhibit.

Unlike assertions, never claims are not tied to specific lines of code but instead describe system-wide properties. Therefore, assertions are best for localised, immediate conditions, while never claims are used to verify complex, system-level properties over time. In this case, we want to state that the Sender cannot generate more than one message before the Receiver receives it, i.e., the currently received message is the one currently generated by the Sender. Formally, `[](received_message == sent_message)`.

To verify this formula in PROMELA, which corresponds to the Flow Control requirement initially specified, we can transform it into an assertion with the instruction `assert(received_message==sent_message)` which should be inserted after the Receiver reads the message from the `receiver_to_sender` channel (i.e., between ln26 and ln27 in Listing 1). This assertion has to be true every time the Receiver reads a message to check that the Sender does not send another message before the Receiver reads the current one. Given the size of the model, Spin is able to verify this assertion using 77 states, 86 transitions, and 128.730 MB, which is mainly used to store the hash table used by the verifier.

The considered property is verified because the model correctly implements a synchronisation protocol between the Sender and the Receiver.

Consider now Listing 2, which is a modification of the previous code in which we eliminate the control channel and the associated lines of code.

```
1  #define MAX_MESSAGES 10
2
3  chan communication = [MAX_MESSAGES] of
       {int};
4  int sent_message = 0;
5  int received_message = 0;
6
7  proctype Sender(chan sender_to_receiver) {
8      int counter = 0;
9      bool x;
10     do
11     :: counter < MAX_MESSAGES ->
12         sent_message=counter;
13         sender_to_receiver!sent_message;
14         counter++;
15     ::counter == MAX_MESSAGES -> goto done;
16     od;
17     done: skip;
18 }
19
20 proctype Receiver(chan receiver_to_sender) {
21 int expected_counter = 0;
22 do
23     ::expected_counter < MAX_MESSAGES ->
24         receiver_to_sender?
               received_message;
25         assert(received_message
               ==sent_message);
26         expected_counter++;
27     ::expected_counter == MAX_MESSAGES ->
          break;
28 od;
29 }
30
31 init {
32     run Sender(communication);
33     run Receiver(communication);
34 }
```

Listing 2: PROMELA code for Sender and Receiver processes without control channel, and with assertion.

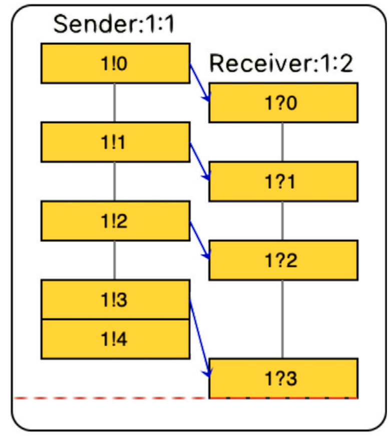

Fig. 1. Counterexample generated by Spin when the synchronisation between processes is not in place.

In this version, the Sender and Receiver communicate without using a control channel. Since the only communication channel (`communication`) can buffer up to `MAX_MESSAGES` messages (ln 1), and the Receiver does not have to wait for a synchronisation signal, then the Sender can continuously send (and buffer) messages (ln 11–13). The Receiver can read a message at any moment it is available (ln 24), but because the Sender is not bound to wait for the Receiver to read before generating the new message, the Sender could generate two or more messages before the Receiver reads them. For this reason, in this case, the property does not hold. Spin can verify it using 52 states (and using 128.730 MB) and identifying a counterexample. The counterexample is presented as the message sequence chart in Fig. 1 where the first line represents the Sender and its actions and the second line represents the Receiver and its actions. The arrows between the two show when a message is sent from one to the other through a communication channel. In the example, given the lack of synchronisation and the size of the communication channel, the Sender can generate a new message (1!4 in the figure) before the Receiver processes an old message (1?3 in the figure).

Once the model has been verified to be correct against all the requirements (in our case, the initially generated model), one can implement the associated code, e.g., in Python, as illustrated in Listings 3 and 4. The Sender first connects to the Receiver (ln 5 in Listing 3) and starts sending ordered messages (ln 11). After sending each message, it waits for an `ACK` (ln 15). If this is not received within two seconds, or if a corrupted message is received, the Sender interrupts the procedure and closes the connection (ln 17–18, ln 20–22, ln 26).

```
1  def start_sender():
2      sender_socket =
           socket.socket(socket.AF_INET,
           socket.SOCK_STREAM)
3      host = 'localhost'
4      port = 12345
5      sender_socket.connect((host, port))
6
7      timeout = 2  # Timeout for waiting for
           ACK (in seconds)
8
9      for counter in range(1, 11):
10         message = 'Message'
11         sender_socket.send(f'{message}
               {counter}'.encode('utf-8'))
12         sender_socket.settimeout(timeout)
13
```

```
        try:
            ack = sender_socket.recv(1024).decode('utf-8')
            if ack != 'ACK':
                print(f'Error: Received {ack} instead of ACK. Stopping Sender.')
                break
            print(f'Received {ack} for {message} {counter}')
        except socket.timeout:
            print(f'Timeout: No ACK received for {message}. Stopping Sender.')
            break

        time.sleep(1)  # Simulate delay between messages

    sender_socket.close()
```

Listing 3: Python code for Sender

The Receiver in Listing 4 starts listening for incoming connections (a maximum of 5 Senders can be connected at the same time, ln 6). After a connection request has been received, this is accepted (ln 9), and the Receiver starts decoding the data (ln 15), which includes a message and a counter (ln 19), which identifies the order of the message. If the counter is different from the expected one, the Receiver will send an `NACK` message and will close the connection (ln 26, ln 32–33).

```
1  def start_receiver():
2      receiver_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3      host = 'localhost'
4      port = 12345
5      receiver_socket.bind((host, port))
6      receiver_socket.listen(5)
7
8      print(f'Receiver started! Listening on {host}:{port}')
9      sender_socket, addr = receiver_socket.accept()
10     print(f'Got a connection from {addr}')
11
12     expected_counter = 1
13
14     while True:
15         data = sender_socket.recv(1024).decode('utf-8')
16         if not data:
17             break
18
19         message, counter = data.split()
20
21         counter = int(counter)
22         print(f'Received message: {message}, Counter: {counter}')
23
24         if counter != expected_counter:
25             print(f'Error: Expected counter {expected_counter}, but received {counter}. Stopping Receiver.')
26             sender_socket.send('NACK'.encode('utf-8'))
27             break
28
29         sender_socket.send('ACK'.encode('utf-8'))
30         expected_counter += 1
31
32     sender_socket.close()
33     receiver_socket.close()
```

Listing 4: Python code for Receiver

We note that some details of the code (e.g., host, port, socket) did not appear in the original model, although the abstract behaviour is the same. For this reason, additional formal verification may be required. Phyton code can be verified directly by Dafny [106], a programming language and verification tool designed to enable formal verification of program correctness by incorporating specifications directly into the code. While Dafny is not natively designed to verify Python code, critical sections of Python code can be translated into Dafny, specifying the required properties and using Dafny to perform the verification. Phyton programs can also be directly analysed using static analysis with tools such as Nagini [112] that can verify memory safety, functional properties, termination, deadlock freedom, and input/output behaviour.

4. Roadmap: Using LLMs to support FM-based development

The example of Section 3 gives an idea of what could be a typical formal RE-based development context, in which formal logics are used to express requirements, formal models define the specification, and formal tools verify the specification against its requirements. We argue that the different steps of the process can be supported by LLMs, and their application can facilitate the uptake of FMs in practice, seamlessly integrating them into standard development processes. For example, LLMs can replace the human in the translation from the model in Listing 1 into the code of Listings 3 and 4. Analogously, LLMs can translate the different requirements into Spin `assert` (as used in the example for the Flow Control requirement), or Spin never claims for global requirements or liveness properties. In the following, we discuss different possible usages of LLMs in a formal RE-based development process, and discuss research avenues, also summarising the main action points (highlighted with 💡). Fig. 2 summarises and connects the different discussion topics presented in the rest of this section. Each discussion topic is associated with a circled number, which also appears in the figure. We will describe the figure at the end of the section to provide a comprehensive view of the roadmap.

Generating FM and SE Artifacts 1 In a development context in which one first develops a formal specification and then implements the software, LLMs can provide support for code generation from formal specifications. One can assume that this task can be easily addressed by LLMs, considering their ability in code synthesis—see, e.g., Code Llama [113] or StarCoder [114]. A specification is more abstract than the desired code—e.g., including specific library functions to create and connect sockets, as in Listings 3 and 4—and one wishes to identify the correct project-dependent or machine-dependent function calls to be included in the code.

To generate code from a formal specification identifying the correct source code functions can, in principle, be achieved through RAG approaches, in which an existing knowledge base is exploited to augment a prompt with information to be used to better inform the LLM. In this case, the knowledge base would be composed of the existing libraries that can be used in the program to be synthesised. This is the approach adopted by the recent preliminary contribution by Kozioklek et al. [115], where the authors use RAG to enrich generated code with proprietary and well-tested function blocks in the field of control systems.

💡 **Action Point:** To facilitate code generation from formal specifications, RAG can be employed. By enriching prompts with code from existing libraries, the generated output can integrate these resources.

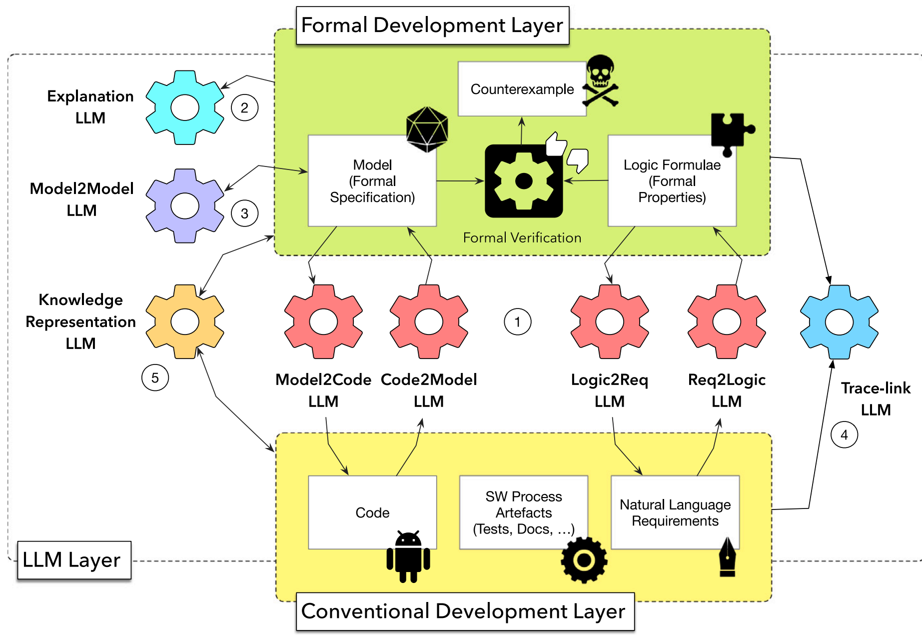

Fig. 2. Overall view of the salient points of the roadmap on using LLM agents to support FM-based development.

Another natural usage of LLMs is the generation of formal specifications and logic formulae based on code and requirements, respectively. Considering our example in Section 3, this means starting from the source code in Listing 3 and 4 to produce the model in Listing 1, or starting from the initial requirements to produce asserts or never claims. The challenge here is the opposite one discussed for specification-to-code translation. Since the source code is typically more refined than a formal specification, the mechanism of translation of source code into a specification needs to be able to abstract from implementation details. This is also needed to ensure that formal verification is feasible with the given memory resources and time constraints, thus avoiding the state-space explosion problem, which typically occurs when performing model-checking of large specifications. Automatically identifying the relevant backbone of a code module or a complex program can be challenging. LLMs have demonstrated a strong ability in summarising code [116], which means generating short functional descriptions of code snippets. This suggests a strong potential in abstracting the meaning of source code, which can, in principle, be exploited also to support the creation of formal specifications.

While works exist that investigate the performance LLM for generating graphical models from NL requirements [9], we are not aware of studies targeting the generation of formal high-level models similar to the one in Listing 1 from source code. It is worth noting that software model checking [117] is a technique that allows creating abstract models from source code and verifying *implementation-related* properties, e.g., memory-safety, array bounds, and runtime errors in general. Instead, traditional model checking, as the one exemplified in Section 3, considers a more high-level model to verify other types of *conceptual* properties, e.g., safety, liveness, and concurrency. Generating these types of high-level models could be beneficial to ensure that the code verifies conceptual properties. We argue that this could be an interesting avenue of research, which could facilitate both the development and maintenance of formal specifications, and the support of legacy systems that often lack specifications and documentation. LLMs may have limited knowledge of formal languages due to the modest FM data used during their training. However, the similarity between some formal languages and certain programming languages (e.g., the similarity between C code and PROMELA code) could facilitate the translation.

Action Point: To generate formal specifications from code or requirements, we can use LLMs summarisation capabilities to abstract from source code and extract essential functional descriptions for formal models. Similarities between programming languages and formal languages can overcome the problem of limited FM knowledge of LLMs.

Concerning the generation of logic formulae from NL requirements using LLMs, Cosler et al. [118] recently introduced `nl2spec`, a tool for translating NL expressions into LTL formulae. The tool addresses the inherent ambiguity of NL through human interaction. Besides the formula, it also generates translations of sub-expressions of the original one, which helps the user identify possible mistakes and apply corrections. However, LTL is only one of the many temporal logics that cannot express all the types of real-world requirements, and more effort is needed to have a comprehensive environment for requirements translation. Initiatives such as the Natural2CTL dataset [119], which includes pairs of NL properties and their translation into Computation Tree Logic (CTL), can be particularly beneficial for the community, as they can be used to fine-tune LLMs and test their performance on the translation task. The translation of NL into logic, independently from the chosen type of logic, can be achieved by first addressing the ambiguity of NL and then generating the equivalent formula. This can be coupled with providing explanations to enable the identification of possible mistakes, as well as the rationale behind the translation.

Action Point: To translate NL requirements into logic, we need to: (1) provide tools that iterate with users to reduce NL ambiguity; (2) provide and leverage specialised datasets with requirement-formula pairs; and (3) enrich translations with explanations.

Explaining FM Artifacts 2 The formal model illustrated in the example, as well as the LTL formula, are rather simple to understand without knowing all the nuances of formal languages and logic. However, formal requirements and specifications can easily become more complex and thus hard to interpret. This occurs especially when real-world systems are modelled and when complex properties need to be expressed, possibly involving real-time constraints, e.g., using Real-time CTL [120], or recursion/iterations, e.g., using μ-calculus [121]. The reader is invited to look at the following repositories, which collect formal specifications and formulae in different formal languages.[6][7][8] [122–124]. Some of the specifications are single files with over 2 K lines of code that could benefit from an explanation, even for someone who is highly familiar with the adopted formal languages. This problem is further exacerbated by the lack of support for modularity of some of the languages [125]. Explainability is one of the ten principles that an FM needs to satisfy to be applicable, according to Gleirscher et al. [126]. In fact, domain experts from companies who wish to use FMs do not typically need to develop formal specifications themselves and normally rely on external consultants [127]. It is thus important that they can understand the specifications to ensure that the information about the system to be modelled, verified, and developed is correctly conveyed. Explainability is also an important feature for the FM experts themselves, especially when dealing with counterexamples generated by model checkers. These can be rather complex, and some engines offer simulation capabilities to execute the counterexamples [128] and visualise the traces that violate a certain property. However, these simulations could also benefit from NL language explanations that describe, e.g., the transitions activated in a model or the statements executed.

LLMs can be used to automatically provide NL explanations for models, formulae, and counterexamples. Again considering our example in Section 3, this could entail: including a summary description of the model, as well as appropriate comments; explain the `assert` (e.g., "The currently received message is the one currently generated by the sender. This prevents overwhelming the receiver".); explain the counterexample in Fig. 1 (e.g., "The Sender can generate more than a message (message 2 and 3) when the Receiver is still processing an old message (message 1)"). A peculiarity of formal languages is that these can have different levels of formality, e.g., formal, such as timed automata, or semi-formal, such as UML. The former can be hard to understand, while the latter can lead to different interpretations. In both cases, NL explanations become crucial. Approaches for code comments generation have been recently developed [129], and we argue that similar solutions could also apply to formal specifications. The idea can be further evolved by taking inspiration from LLM-based solutions for code explanation, such as the one from Nam et al. [130]. This method enhances code comprehension by providing localised textual illustrations of the code content directly within the development environment. Concerning explanations of formulae, while works have been performed to translate NL requirements into formal logics [118], tools that explain formulae to non-experts using LLMs are lacking. To our knowledge, preliminary studies have been performed only using more traditional deep learning approaches for machine translation [131]. Concerning counterexamples, these can be particularly complex and hard to follow. However, these are not too dissimilar from stack traces generated by faulty code. Preliminary approaches for explaining stack traces via LLMs have been recently developed [132], and similar solutions can also be applied to support the navigation of counterexamples through the simulation of faulty formal specifications.

Action Point: Explanations of FM artifacts can be achieved by leveraging techniques from code comment generation, localised textual illustrations, and stack trace analysis.

Translating Formal Languages (3) As noticed by previous work, each formal language and environment has its own unique peculiarities [127, 133]. Some focus on quantitative requirements (e.g., performance, availability), some on qualitative ones (e.g., safety, security); some provide friendly GUIs and visual simulation, some others offer support for code generation, etc.—see [134] for a systematic evaluation of 13 formal tools. A formal RE-based development process should be able to integrate different languages and tools to address the diverse dimensions of the system to be verified, and provide models that target different audiences, e.g., Simulink models for practitioners *vs* Spin models for FM experts. This was also observed in [135], where Real-Time Graphical Interval Logic templates are compositionally encoded in different temporal logics, thus enabling the portability of specifications. To this end, FM practitioners should be able to automatically translate specifications from one language to another to guarantee consistency between the models used by the tools, or to provide different views of the system—for example, by omitting or adding details to achieve different levels of abstraction—even when using the same source and target language. Consistency between models is important to ensure FM diversity [136], which provides further guarantees of the soundness of the verification process, especially in the presence of tools that have not been certified and can thus generate incorrect output despite their formal foundations. LLMs have demonstrated abilities in code translation for different programming languages [137]. This code translation technique can be fruitfully exploited also in a scenario in which different FMs are adopted in system development to address different concerns and verify multiple types of properties.

Action Point: To translate formal languages into one another, thus adapting models to different verification environments and audiences, the code-to-code translation capabilities of LLMs can be exploited.

Supporting Iterations and Evolution (4) The simple example provided assumes that starting from requirements, one passes to models and formulae to be verified and then to code. However, like any system development process, a formal RE-based one is also incremental and iterative. One can start from early NL requirements or even high-level system descriptions and then provide specifications, which are then refined, together with the requirements, and eventually translated into code. The code and the system as a whole—possibly including hardware and communication components—go through different levels of testing (unit, integration, acceptance, etc.), possibly through the support of FMs for test generation such as model-based testing [138]. In a rigorous development process, this creates incremental versions of requirements documents, specifications, tests, and other process artefacts, which need to be kept aligned with trace-links to ensure consistency across artefacts. Trace-link identification has been a long-standing NLP problem [139]. Given the ability of LLMs to interpret both NL expressions and code, we argue that a combination of code-specific LLMs and NL-oriented ones can be beneficial in taking a step forward in addressing this challenging problem. Early studies in this direction have already been performed by Rodriguez et al. [6].

Action Point: To support iterations and evolution, we can leverage a combination of code-specific and NL-oriented LLMs to enhance trace-link identification and ensure consistency across all artefacts produced in the development process.

Automating Knowledge Engineering (5) Knowledge engineering is the collection of activities for eliciting, capturing, conceptualising, and formalising knowledge to be used in information systems. It provides support for formal RE activities, especially in creating and analysing models, and typically utilises formal or semi-formal representation languages. As noticed in [140], LLMs can significantly streamline the process of extracting and structuring knowledge from unstructured data sources. By leveraging LLMs, systems could autonomously identify, extract, and integrate relevant information, reducing the reliance on manual input and expertise. Analogously, Mateiu and Groza [141] emphasise the role of LLMs in ontology engineering, where these models can aid in creating, updating, and refining ontologies. This supports more dynamic and up-to-date knowledge bases. LLMs also have the potential to improve the semantic understanding of systems [142] by enhancing the ability of systems to understand and reason about complex, context-dependent information, which is crucial

[6] https://github.com/hhu-stups/specifications Last visit 22 Oct. 2024.
[7] https://www.rers-challenge.org/2018/. Last visit 22 Oct. 2024.
[8] https://qcomp.org/benchmarks/. Last visit 22 Oct. 2024.

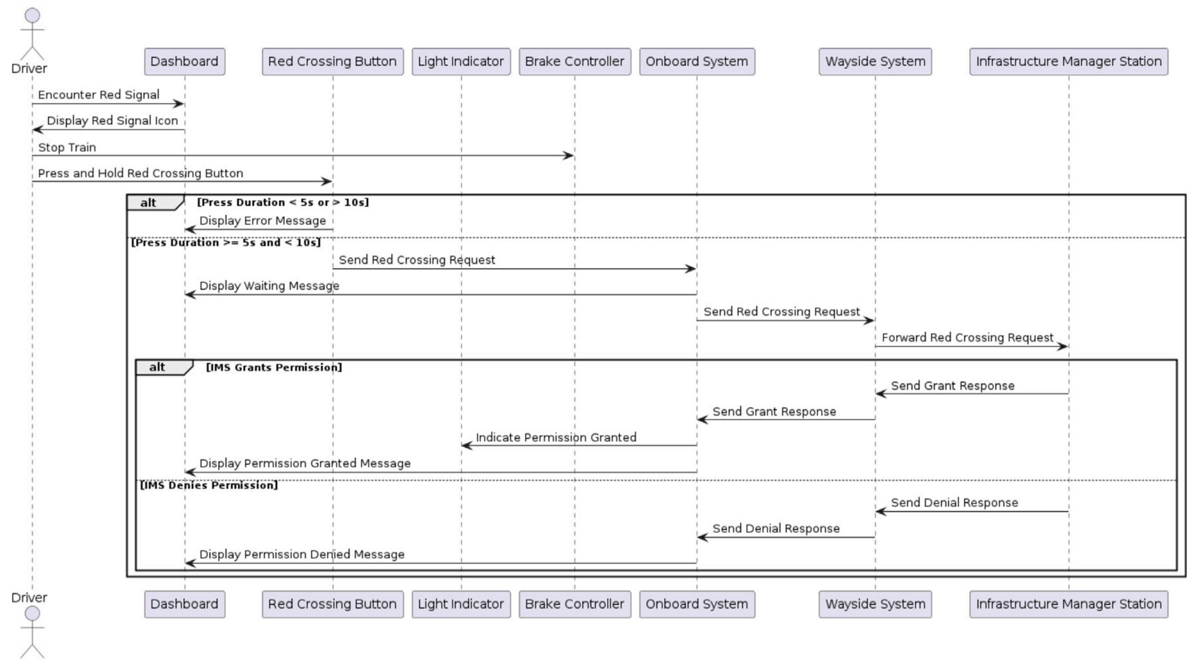

Fig. 3. Generated UML sequence Diagram for the red-crossing function.

for effective knowledge representation and reasoning. In the example of Section 3, knowledge can be extracted from the different artefacts, such as requirements and models, as well as other process artefacts, e.g., documentation and tests. This knowledge can be exploited to build ontologies. These can be used, e.g., to check the consistency between artefacts or inform the different LLMs that produce artefacts, explanations, and traces.

Action Point: To automate knowledge engineering, LLMs can be used to automatically create ontologies from, possibly unstructured, software engineering artefacts. Ontologies can in turn be used to support LLM-driven development (cf. Section 5).

Summary The vision, summarised in Fig. 2, is structured into three interconnected layers,[9] a formal development layer, a conventional development layer and an LLM layer, which is composed by different LLM agents. In the original formal development process exemplified in Section 3, the interaction between the two layers is manual, i.e., one needs to manually translate models and logic formulae from/to code and requirements, respectively. In the envisioned process, the interaction is delegated to LLMs. In particular, conventional and formal development layers exploit different LLMs for translating conventional artefacts into formal ones, and vice versa (1). Formal artefacts can be explained with LLMs (2), while other LLMs can provide model-to-model transformations (3). The LLM layer can also contribute by creating trace links between all the artefacts of the two layers, thus supporting artefact evolution (4). Finally, process and domain knowledge can be extracted from the different artefacts through LLMs, and can be further exploited by the different development layers and LLMs (5).

5. LLM-driven software development: an example

LLMs can be used to support the RE and SE process, and different works have been published in this sense, mainly focusing on code activities such as code generation and testing [2,3]. In the following, we exemplify a possible RE-centred development process in which requirements can be generated from high-level descriptions, analysed for ambiguity and completeness, improved, classified, and traced with other requirements. LLMs can also be used to automatically generate graphical models, code, comments, and tests. Once an app has been released, user feedback can be analysed to improve the requirements.

In the following, we exemplify prompts and associated LLM output to implement a process composed of the steps outlined above. To generate the output, we have prompted the online interface of ChatGPT 3.5 and slightly adjusted the results to make them more synthetic. We want to remark that the adjustments were limited, although some iterative prompting was required in some cases. In particular, we reduced the verbose replies typical of ChatGPT, and had to iterate a few times before obtaining a clear and faithful diagram such as the one in Fig. 3, prompting the LLM to add or remove details. To make our storytelling more natural, we will not strictly follow the sequence of steps. Furthermore, we will consider different types of systems, as not all tasks are appropriate for all systems.

Requirements Generation. Let us start with a prompt that generates creative requirements for an enhanced guitar tuning mobile app: *Please act as a creative requirements engineer. Could you please provide a set of user stories for an app that should support guitar tuning and also embed other social functionalities?* An excerpt of the LLM output is:

[9] Here, layers should be intended as graphical layers, similar to those in Photoshop, i.e., distinct groups of elements that exist at different depths within the composition, with some forming the foreground (Formal Development Layer and Conventional Development Layer) and others receding into the background (LLM Layer). This applies also to Fig. 4.

- R1: As a guitarist, I want to use the app to accurately tune my guitar by providing visual indicators and audio feedback, so I can achieve precise tuning for each string.
- R2: As a beginner guitarist, I want the app to offer step-by-step instructions and tutorials on guitar tuning, helping me learn and improve my tuning skills.
- R3: As a guitarist in a band, I want the app to allow me to save and recall custom tunings for different songs or playing styles, so I can quickly switch between tunings during rehearsals or performances.
- R4: As a musician, I want the app to provide a variety of alternate tunings for different genres or playing techniques, allowing me to explore new sounds and experiment with different tunings.
- R5: As a guitarist interested in learning new songs, I want the app to provide a library of popular songs with their associated tunings, enabling me to tune my guitar correctly for each song I want to learn.
- R6: As a guitarist who enjoys jamming with others, I want the app to include a jam session feature where I can connect with fellow musicians, share recordings, and collaborate on improvisations or song ideas.
- ...

We observe that the output includes both typical user stories that one can expect from any guitar tuning app (e.g., R1) as well as creative solutions for interacting with other musicians (e.g., R6).

User Feedback Analysis Let us now imagine that we have developed the app, we have released it to the public, and we want to summarise the feedback from users. We can imagine using a RAG system to retrieve users' feedback and trigger the LLM with the following prompt: *Please act as an expert in summarising user feedback. What are the main concerns of the users? Please associate each concern with a non-functional requirement class.*

- **Accuracy:** Some users complained that the app is not accurate enough for tuning their guitars, especially in noisy environments or with alternate tunings. This could affect the usability and reliability of the app.
- **Performance:** Some users reported that the app is slow to load, crashes frequently, or drains the battery too fast. This could affect the efficiency and availability of the app.
- **Security:** Some users expressed concerns about the app's data collection and privacy policies, especially regarding microphone access and location tracking. This could affect the confidentiality and integrity of the app.
- **Compatibility:** Some users mentioned that the app is not compatible with their devices, operating systems, or other apps. This could affect the portability and interoperability of the app.
- **Cost:** Some users felt that the app is too expensive for its features, or that the subscription model is not fair or transparent. This could affect the affordability and maintainability of the app.

Smell Detection In the development process of mission-critical systems, it is typical to perform requirements reviews to identify ambiguities and smells in general. Let us consider the following prompt: *Please act as an ambiguity detection tool for requirements. Does the following requirement contain any ambiguities? **Requirement:** The onboard device sends a connection request to the wayside and monitoring subsystem, and it responds with an acknowledgement or aborts the connection.*

The requirement contains a potential ambiguity:

It responds with an acknowledgment or aborts the connection: The ambiguity here lies in what exactly is meant by "it". Does "it" refer to the onboard device, the wayside and monitoring subsystem, or some other entity? It's unclear which component is responsible for responding with the acknowledgment or abort the action.

To clarify this requirement, you might want to specify which entity is responsible for sending the acknowledgment or initiating the connection termination. For example, "The onboard device sends a connection request to the wayside and monitoring subsystem. The subsystem responds with an acknowledgment if successful, or it initiates an abort of the connection request if necessary". This clarification makes it clear which entity is performing each action.

The LLM identifies a so-called *anaphoric ambiguity*, due to a pronoun for which it is unclear which is the referent. Not always anaphorae lead to ambiguity, and the literature distinguishes between *nocuous* and *innocuous* ambiguities [143]. The former are the ones for which a reader can easily find the referent, while the latter are the ones that may cause misunderstandings. LLMs are expected to have a deep understanding of the semantics of a text and can show the capability to recognise innocuous ambiguities. Let us consider the following prompt, including a rephrasing of the previous requirement: *Please act as an ambiguity detection tool. Is this requirement ambiguous? **Requirement:** When the system receives a corrupted message, it terminates the connection with an abort message.*

The requirement seems relatively clear and straightforward, but there is a minor potential ambiguity that could be clarified:

"Corrupted message": The term could be interpreted in multiple ways. Does it refer to a message with invalid syntax, a message with incorrect data, or a message that has been tampered with during transmission? Providing a precise definition or clarification of what constitutes a "corrupted message" would help eliminate any ambiguity.

In this case, the pronoun is not recognised as a source of ambiguity, as the referent can clearly be inferred from the context (it is an *innocuous ambiguity*). On the other hand, the ambiguity detection LLM is able to recognise a case of generality, i.e., an expression that is too abstract and can be interpreted in different ways. Similar to this case, one can analyse other requirements defects, such as inconsistency and incompleteness. This second case is demonstrated in the following.

Completeness Check and Requirements Completion Before proceeding with development, it is important to ensure that requirements are sufficiently complete. In this case, let us consider a larger set of requirements for a railway subsystem function that enables the so-called procedure of red-crossing, i.e., a procedure that allows the driver to cross a red signal after receiving authorisation from the infrastructure manager. The prompt is reported in the following framed box.

I have the following requirements for a railway system in relation to red signal crossing permission. Are the requirements that I provided sufficiently complete to make sense of the system behavior? Please consider incompleteness both at the requirement level (incompleteness of a single requirement) and incompleteness of the requirements set (in case requirements are missing).

Requirements:

1. When the driver encounters a red signal, the driver shall stop the train.
2. After the driver stops, the driver can either wait for a green signal or press the Red Crossing button and perform the Red Crossing Procedure.
3. If the driver decides to perform the Red Crossing procedure, it first needs to receive authorisation from the infrastructure manager.
4. Then, after the authorisation, the driver can press the Red Crossing button for at least 5 s and less than 10 s.
5. After 5 s, a light blinks on the panel.
6. When the light blinks, the driver can perform the red crossing.
7. The driver shall perform the red crossing within 30 s after the light blinks.
8. If the red signal is not crossed within the 30 s, the onboard system shall automatically stop the train.
9. After 30 s from the moment in which it was lighted on, the light stops blinking.

The generated output is:

The requirements that you provided **are not sufficiently complete** to make sense of the system's behaviour. There are some gaps that need to be resolved. Here are some ways to modify the requirements to complete them:

At the requirement level, you should specify the conditions and actions more clearly and consistently. For example, in requirement 2, you should state what happens if the driver does not press the red-crossing button or does not perform the red-crossing procedure. In requirement 4, you should state what happens if the driver presses the red crossing button for less than 5 s or more than 10 s. In requirement 6, you should state how the driver performs the red crossing (e.g., by pressing a pedal or a lever).

At the requirements set level, you should add some requirements that are missing or implied by the existing ones. For example, you should state what is the purpose and scope of the system, what are the roles and responsibilities of the driver and the infrastructure manager, what are the assumptions and constraints of the system, what are the expected outcomes and quality attributes of the system, and what are the exceptions and error handling mechanisms of the system.

We can then ask the LLM to complete the requirements. Prompt: *Please rewrite the requirements so that they can be considered complete.*

- **Architectural Requirements**
  - (a) The system shall provide a red signal crossing permission function for the driver of the train when the train encounters a red signal on the railway track.
  - (b) The system shall consist of an onboard system and a wayside system that communicate with each other via wireless signals.
  - (c) The onboard system shall consist of a dashboard, a red crossing button, a light indicator, and a brake controller.
- **Functional Requirements**
  - (1) When the train encounters a red signal, the onboard system shall display a red signal icon on the dashboard, and the driver shall stop the train before the signal.
  - (2) After the driver stops the train, the driver can either wait for a green signal or request a red crossing permission from the infrastructure manager by pressing and holding the red crossing button for at least 5 s and less than 10 s.
  - (3) If the driver presses and holds the red crossing button for less than 5 s or more than 10 s, the onboard system shall display an error message on the dashboard and ignore the request.
  - (4) If the driver presses and holds the red crossing button for at least 5 s and less than 10 s, the onboard system shall send a request message to the wayside system and display a waiting message on the dashboard.
  - (5) The wayside system shall receive the request message from the onboard system and forward it to the infrastructure manager station.
  - (6) The infrastructure manager shall review the request message and decide whether to grant or deny the red crossing permission based on the safety and operational conditions of the railway track.
  - (7) The infrastructure manager shall send a response message to the wayside system indicating the decision and the reason.

The LLM not only creates a list of requirements that better specify the expected behaviour but also includes architectural requirements, as well as the requirements for the infrastructure manager (the last two requirements), which were not considered in the initial requirements set.

Model Generation When requirements have been defined, one passes to the system design, which frequently leverages the support of graphical representations, such as UML diagrams. Let us consider the red-crossing function that we already used to demonstrate the activity of completeness check and requirements completion, and let us generate a UML sequence diagram. When dealing with non-multimodal LLMs, i.e., models that can only generate texts and not graphical representations, it can be useful to resort to the PlantUML structured language and the tool PlantText.[10] This is a tool that interprets the language and visualises the model, as done by Ferrari et al. [9]. Prompt: *Please act as a requirements engineer expert in system design. Generate a UML sequence diagram in*

[10] https://www.planttext.com/ Last visit 30 Oct. 2024.

PlantUML so that it can be visualised with PlantText. Requirements: {...}. The requirements used are the same as those illustrated for the red-crossing function after their completion, and the output is depicted in Fig. 3.

Requirements Classification After requirements have been analysed, one can imagine classifying them into different nonfunctional categories to facilitate structured requirements documentation and requirements storage for reuse. Requirements classification is one of the most common tasks in the NLP for RE field [31]. Prompt: *Please act as a requirements classification tool. What are the possible nonfunctional requirements classes of this requirement? "The authentication system should be easy to use and prevent access from underage users".*

> The requirement can be classified into the following nonfunctional requirements classes:
>
> 1. **Usability or User Experience:** This class encompasses the aspect of the requirement that focuses on the system being "easy to use". It pertains to the user's interaction with the authentication system, ensuring that it is intuitive and user-friendly and minimises the cognitive effort required for users to authenticate themselves successfully.
> 2. **Security:** This class covers the aspect of the requirement that emphasises the system's ability to "prevent access from underage users". It deals with security-related concerns, including access control mechanisms, user authentication processes, and measures to ensure that only authorised users who meet certain criteria, like being of a certain age, can access the system.

It is interesting to notice that the LLM is able to actually identify two possible classes, which underlines its ability to have a clear comprehension of the nuances of the requirement statement. A requirements classification system can be foreseen in which one inputs a set of possible classes, and the LLM produces a less verbose output where each requirement is associated with its class, as done in previous work [37].

Requirements Tracing Another typical activity in mission-critical RE processes is requirements tracing, which enables the identification of trace relations between different requirements and between requirements and other process artifacts, e.g., models, code, and tests. Let us consider the problem of requirements-to-requirements tracing with LLMs: *Please act as a requirements tracing tool. Shall these two requirements be traced to each other? "1. the onboard system shall have a dashboard". "2. the onboard system shall notify the driver of the maximum speed through an electronic speedometer".*

> **Yes, these two requirements should be traced to each other.** They are highly related because they belong to the same system and have a common function. The onboard system is responsible for displaying information to the driver and passengers, and one of the information is the maximum speed. **The electronic speedometer is a component of the dashboard that shows the maximum speed to the driver.** Therefore, there is a dependency and a consistency between these two requirements, and they should be traced to each other.

It is worth noting that the LLM is able to infer that the speedometer is actually part of the dashboard, although this is not explicitly stated in the second requirement.

Code, Comments, Test Generation, and other Code-related Tasks Finally, one can go ahead with the classical development activities with code generation, code documentation through comments, and test generation. Other activities that could be performed with LLMs, and not illustrated in this showcase are code summarisation [144], code revision [145]—see Hou et al. [3] for a comprehensive overview of code activities enhanced by LLMs. Code-centric tasks typically require the usage of LLMs specialised for code tasks, such as CodeLlama [113] or StarCoder [114]. We do not illustrate these code-related tasks, as our paper is mainly focused on LLMs for RE.

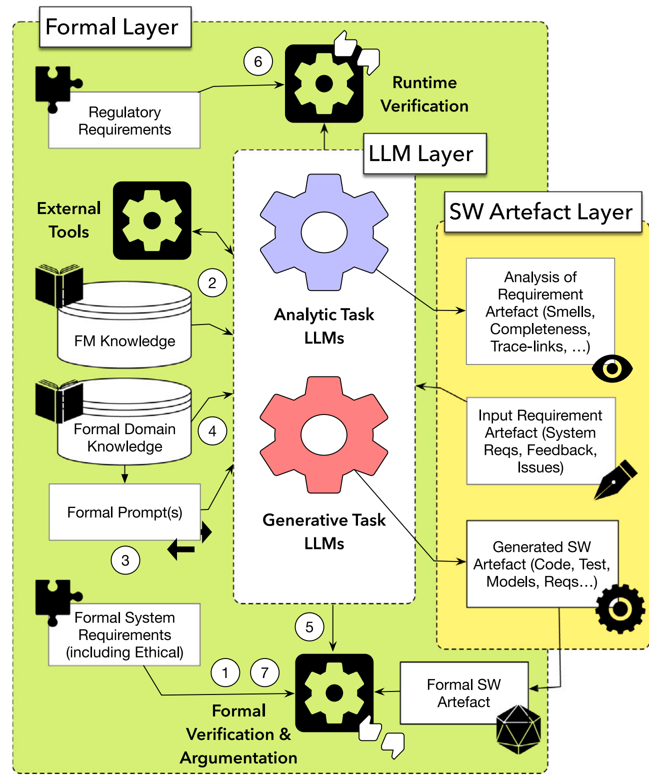

Fig. 4. Overall view of the salient points of the roadmap on using FMs to support LLM-based development.

6. Roadmap: Using FMs to support LLM-based development

The LLM-based software development process exemplified in Section 5 provides a view of how LLMs can be exploited to support different RE and software engineering tasks. The illustrated workflow can be implemented through LLM agents [30], which can perform specific RE and SE tasks, also using external resources, e.g., knowledge bases or external libraries. However, by delegating part of the system development to LLMs, there is a risk of losing control of the process, especially when LLM agents are automatically orchestrated. In our vision, FMs can provide the means to restore this control. In the following, we report research avenues in this direction, again highlighting action points with the symbol 💡. Fig. 4 summarises and connects the different research avenues discussed. Each discussion item is associated with a circled number, which appears in the figure and in the following paragraphs. We will describe the figure at the end of the section to provide a comprehensive view of the roadmap.

Ensuring Correctness through Formal Requirements and Argumentation 1 As well known, LLMs can sometimes generate incorrect solutions to RE problems or "hallucinations". These may not be immediately evident since the output, being it a requirement, a specification, or another RE artifact, may appear sufficiently plausible to the user. We argue that, in the field of RE, addressing this problem is not different from the task of requirements quality assurance, in which analysts review the human-written requirements for ambiguity, inconsistency, and other defects. In principle, quality assurance will need to be applied to LLM-generated artefacts rather than to human-written ones. However, quality assurance through manual review can be tedious and error-prone, and NLP solutions have long been experimented with to facilitate the work of requirements analysts, e.g., by detecting ambiguities, incompleteness, and other defects [31]. In a world where the main NLP solutions for requirements analysis are based on LLMs, we need other means to ensure requirements quality since one cannot fully rely on their output. Moreover, given the plausibility of LLM answers, human analysts and especially novices may tend to rely on the LLM's responses without analysing their truthfulness with a sufficiently critical eye. Formal notations have been explored in the past as one of the solutions to increase requirements quality, as they can ensure that requirements are documented in a way that can unequivocally be interpreted [10]. While their practical application has often been hindered by their inherent complexity, the explanation potential offered by LLMs discussed in Section 4 can in turn help their introduction in the RE quality assurance process. What if the explanation is also faulty? The definition of formal specifications, their verification against the formalised requirements, and the provable consistency of the FM-centred process can mitigate this problem.

Another important support to ensure the correctness of the output can come from formal argumentation theory. Its usage has been explored in RE to support ambiguity identification [146], and represent the conversation between requirements analysts and stakeholders [147]. It has been shown that LLMs possess weak inference abilities and still lag in terms of logical coherence [148], which leads to incorrect output. Constraining the LLM responses to adhere to a certain formal argumentative structure can give evidence of the soundness of the proposed reasoning, thus mitigating hallucinations and incorrect answers.

Action Point: To ensure the correctness of LLM-generated solutions, a formal development process can be adopted, supported by LLM-generated explanations. Moreover, the use of a formal augmentative structure to constrain LLM responses can enhance logical coherence and mitigate hallucinations.

Improving Mathematical Reasoning with Formal LLMs (2) LLMs have demonstrated tremendous abilities when dealing with language-related generative problems, such as text summarisation or question-answering. However, they have also shown difficulties when dealing with mathematical problems, even trivial ones. Requirements for cyber–physical systems often include mathematical formulae [149], used to model physical phenomena, which need to be embedded in the software, e.g., through numerical approximation. These requirements can prove to be challenging, especially when using LLMs to generate code. The lack of mathematical ability of LLMs can, in principle, be due to the lack of mathematical and logic data used during the training process. Creating an LLM using mathematical knowledge has been experimented with in LLMs such as Lemma [150], which has also been used to support theorem proving, and MathStral.[11] However, other authors have highlighted that there may be inherent barriers to the mathematical reasoning of LLMs, since they are not robust to small numerical variations, or modification of logical clauses in their prompts [151]. RAG solutions can provide help in this direction by introducing external knowledge based on formal notions that can be retrieved and used by the LLM when addressing logical, mathematical, or formal problems, as done, e.g., in LeanDojo [152]. Similar approaches can be followed in RE and SE by creating RAG systems that embed FM knowledge. A possible alternative is to create multiple LLMs based on FM knowledge, code knowledge (e.g., CodeLlama [113], StarCoder [114]), and NL knowledge and devise approaches to interrogate them simultaneously and combine their output in a meaningful way. Finally, the LLM agent paradigm [30] can be used to access external libraries and resources, e.g., a reasoner or a calculator.

Action Point: The challenges of using LLMs for generating code that embeds mathematical constraints can be addressed by leveraging specialised training, dynamic knowledge retrieval with RAG, and collaboration between multiple LLM agents embedding different types of knowledge (code, FM, and NL). The agents can also use external resources, e.g., for computation and logic reasoning.

Formal Prompt Engineering (3) With LLM, prompt engineering has emerged as an entirely new discipline, focused on crafting the most appropriate NL query for an LLM to solve the task at hand [153]. When prompting to generate source code or design models, prompts basically become requirements to be fulfilled, and, in this context, RE boils down to prompt engineering. Similar to requirements, NL prompts can suffer from ambiguity issues, which can have consequences on generated artefacts [9], and their expression can benefit from the usage of formal notations or rigorous controlled NLs. In this sense, researchers are encouraged to experiment with the usage of notations such as ACSL (ANSI/ISO C Specification Language), which helps formally specify the expected behaviour of C programs and is used by the FRAMA-C environment [105]. By specifying formal pre- and post-conditions directly in the prompts, one can have the guarantee that the constraints are not only considered during generation but are also embedded in the generated code and can be checked by the analysis environment.

The proposed idea can be effective when one expects to generate single functions, but it does not scale to large software systems, as the specified conditions are typically low-level, i.e., associated with code variables. Instead, when complex software systems need to be generated, requirements analysts need to engineer *architectures* of inter-connected and inter-dependent prompts. The design of this architecture can benefit from classical semi-formal languages such as UML, which allow the representation of relations between components (i.e., prompts, in this context) and guide their constructions and orchestration. Researchers are thus invited to adapt consolidated model-driven development principles and approaches to the novel scenario of prompt engineering, possibly evolving and adapting them to account for the peculiarities of prompt and prompt relations.

Action Point: Using formal notation or controlled natural languages can help enhance the precision of NL prompts. Incorporating formal pre- and post-conditions in prompts can help embed constraints in generated code. For complex systems, we can develop interconnected prompt architectures using semi-formal languages to represent relationships between prompts and guide their orchestration. To this end, an LLM agent-based paradigm [30] can be used.

Formal Domain Knowledge and Explainability (4) Domain knowledge is one of the key dimensions of any RE endeavour, as the interpretation and analysis of the requirements first require a thorough understanding of the domain of the system-to-be, its stakeholders, application context, and constraints. LLMs tend to acquire knowledge from large amounts of unstructured input documents and encapsulate it in the parameters adjusted through the learning process. This has been possible because of the large availability of NL documents on the web, which have been used as a stream of information to train LLMs. However, when dealing with domain- and project-specific knowledge, for which textual information is scarce, LLM cannot be expected to perform as well as for tasks requiring general knowledge only. We argue that, in contexts where domain-specific corpora are limited, knowledge structured through formal ontologies in the form of knowledge graphs shall be injected in an LLM-based RE process, e.g., with the support of RAG approaches. This would address the need for abstraction that is required for reasoning and that LLMs have demonstrated to lack in some cases [154]. As LLMs are trained only on text, they also lack the concept of a *world model*, i.e., "structure-preserving, behaviourally efficacious representations of the entities and processes in the real world" [148,155]. Although multi-modal LLMs are being developed [156], which can potentially address this issue, formal representations of the entities and processes in the real world can address three important problems. First, formal

[11] https://huggingface.co/mistralai/Mathstral-7B-v0.1. Last visit 21 Oct. 2024.

knowledge graphs can help to constrain the reasoning process, thus preventing hallucinations. Secondly, knowledge graphs can provide a compact and parsimonious representation of knowledge, which is needed when textual information about a domain is not sufficiently abundant to train the LLM and can also positively impact efficiency. Finally, an external formal knowledge representation can support the explainability of the output, as the LLM can resort to a knowledge graph not only for retrieving information but also for justifying its content.

Action Point: Injecting domain-specific knowledge using formal ontologies and knowledge graphs can be used to constrain the LLM reasoning process, prevent hallucinations, and improve explainability and justification of the output.

Ensure LLM Output Consistency through Formal Verification 5 AI components in general, and LLM ones in particular, can bring potential benefits even in mission-critical systems, such as those in healthcare, banking, law, and transportation. Systems that provide recommendations to physicians, lawyers, and financial operators based on LLMs have already been designed [157], and one can expect that these will become commonly used in the future. Given the criticality of their role, these systems have to go through rigorous verification and validation processes. The main issue for LLM-based systems compared to traditional systems is their predictability, i.e., once a test has covered a certain behaviour, a traditional system is expected to exhibit the same behaviour in the future, given the same test case. Instead, for LLMs, this cannot be fully ensured. Although one can apply redundant approaches, such as, e.g., self-consistency [158], FM can help address the problem in the verification process itself. In particular, abstract interpretation can be used to approximate the behaviour of the neural network in a way that can be mathematically analysed. This can be done by abstracting the possible behaviour of the different activation functions used by the neural network, e.g., ReLU, Sigmoid, and verifying the consistency of the output in the presence of perturbations of the prompt or other aspects that determine the variable behaviour. Contributions in this direction have been published that apply to different neural network architectures [159]. We argue that similar approaches can also provide benefits for transformer architectures and LLMs. In addition, other formal abstraction methods [160] are emerging, which help omit specific details of a network that are not relevant to a specific feature, thus making their verification feasible. This is particularly important for safety-critical systems using LLM components, as these are expected to have well-defined and restricted functionalities, which also restrict the focus of their verification.

Action Point: Abstraction techniques used to approximate and mathematically analyse neural network behaviour and verify output consistency under prompt perturbations should be applied to the transformer architecture used in LLMs.

Regulatory Compliance at Runtime 6 Regulatory compliance is concerned with the assurance that a system complies with requirements expressed in regulations, e.g., General Data Protection Regulation (GDPR), and it is one of the central topics in RE [161]. In particular, mission-critical systems with LLMs at their core do not only need to exhibit repeatable behaviour but also adhere to regulations. Formal approaches for regulatory compliance have been developed in the past [161,162], and these can become more applicable thanks to the usage of LLMs discussed in Section 4.

However, compliance cannot be ensured once and for all in a context where an LLM evolves based on new knowledge on which it may be fine-tuned, and also laws are subject to change. Even if an LLM behaviour is made repeatable, this does not mean that the new knowledge that can be introduced has no effect on it. In this context, the concept of runtime verification [163], which ensures that certain properties are fulfilled during system evolution, could be adapted to address the needs of an ever-changing regulatory and knowledge context.

Action Point: As LLMs evolve based on new knowledge and regulations change over time, we need to adapt runtime monitoring to continuously ensure regulatory compliance.

Mitigate Bias and Address Ethical Concerns 7 LLMs are subject to many trustworthiness threats [164], including toxicity, stereotype bias, adversarial robustness, out-of-distribution robustness, robustness on adversarial demonstrations, privacy, machine ethics, and fairness. These threats are currently measured and investigated through testing-based approaches, such as the use of bias testing frameworks designed for SE tasks [165]. To address the problem in a more systematic way, in line with the guidelines presented in [166], we envision the formalisation of a set of ethical requirements. These can be validated through formal techniques on LLM-generated artefacts.

Action Point: Strategies for the formalisation of ethical requirements need to be devised, bridging the gap between value-related concepts and the formal operationalisation of such concepts.

Summary The vision, summarised in Fig. 4 is structured into three layers, a formal layer, a software (SW) artefact layer and sa LLM layer. The LLM layer interacts with the SW artefact layer to implement the LLM-based development process exemplified in Section 5. The LLM layer takes as input requirements artefacts, including traditional forms of system requirements—e.g., "shall" requirements or user stories—as well as other forms, such as user feedback or issues from issue tracking systems.[12] Based on these artefacts, two types of tasks can be carried out by the LLMs: (i) *analytic tasks*, i.e., tasks analysing the input and producing annotations, e.g., smells, tace-links, completeness issues; (ii) *generative tasks*, i.e., tasks that produce other artefacts, e.g., code, models, and even other requirements. These different types of LLM-supported tasks have been exemplified in Section 5. The formal layer enters into play to support the LLM-based development process. In particular, formalised system requirements can be verified against formal SW artefacts associated with LLM-generated ones 1. FM and logical knowledge can be used for training LLMs, and external tools, e.g., calculators and reasoners can be accessed to enhance LLM capabilities 2. Formal prompts can be used to better guide and orchestrate the generative capabilities of LLM agents 3. Formal domain knowledge, e.g., through formal ontologies, can support the generation and analysis of domain- and project-specific artefacts and can foster explainability 4. Formal verification can also be used to verify LLMs, especially in the case of mission-critical systems 5. Runtime verification can be used to facilitate the monitoring of regulatory requirements in the presence of LLM evolution triggered by new knowledge 6. Finally, formal system artefacts can be verified against formalised ethical requirements 7.

7. Practical considerations and limitations

The two roadmaps outlined offer a vision of how FMs and LLMs could be integrated into future software development processes. The vision is unavoidably non-exhaustive, and the actual implementation of the different research directions requires careful planning to prevent risks and address limitations. In the following, we outline practical considerations regarding the implementation of the roadmaps, as well as possible barriers to its unfolding.

Collaboration Between LLM and FM Experts LLMs and FMs are highly specialised fields that require significant expertise to fully master their technical intricacies and utilise their tools effectively. Successful implementation of the envisioned roadmaps also depends on collaboration and clear communication between LLM experts and FM experts. These specialists, though, often come from different schools of thought (statistical vs deterministic thinking), making interdisciplinary collaboration challenging. Facilitating this cooperation falls within the domain of requirements engineers, who play a key role in bridging the gap between these diverse fields. A similar observation was made by Franch et al. [167], when speaking about the role of requirements engineers in

[12] The LLM layer can also take other SW artefacts as input, but here we mainly focus on a RE-centred process, to simplify the figure and the narrative.

the development of systems with artificial intelligence (AI) components at their core.

Empirical Evaluation As in any type of research endeavour, empirical evaluation is crucial to assess the effectiveness and reliability of the approaches being developed. The use of generative AI solutions like LLMs poses challenges in RE due to the lack of datasets for evaluation. This issue becomes even more pronounced when FMs are also involved, as the empirical maturity of the FM field remains limited, particularly when considering their application in practical settings [168]. Furthermore, some generative capabilities of LLMs cannot be evaluated using pre-defined ground truths. For instance, it is not feasible to assume that there is a single correct formal specification for a given piece of code, or vice versa. In such cases, qualitative evaluation methods, typically employed in interview studies or case studies—such as thematic analysis [169] or grounded theory [170]—should be incorporated, as also done by Ferrari et al. [9]. With these methods, one can qualitatively yet scientifically check the output of the LLMs, and assess its quality in a critical way without relying on a pre-defined ground truth. Facilitating the acceptance of these research methods in fields that typically rely on quantitative evaluation approaches, such as LLMs, or adhere strictly to positivist principles, such as FMs, remains a significant hurdle to overcome.

Overreliance on LLM Output LLMs can notoriously hallucinate and produce incorrect outputs in response to a given prompt [171]. However, the generated output often appears plausible, and the verbose explanations provided by LLMs, such as ChatGPT, can convey a sense of machine self-confidence that may mislead a requirements engineer. As a result, there is a risk of over-reliance on the generated content. While the integration of FMs in the software process aims to mitigate this issue with formal prompts and formal verification, additional measures are necessary to ensure that users do not blindly trust the output of LLMs. These include but are not limited to: (a) the definition of novel human-centred quality control mechanisms; (b) the identification and prevention of typical hallucinatory patterns; and (c) development processes specifically tailored for LLM development and training. These are expected to test the LLMs' robustness before they are deployed in software development toolchains such as the ones envisioned in this paper (cf. Fig. 2 and 4).

Diminishing Role of Human Creativity In the figures of our roadmaps, we deliberately excluded human actors to emphasise the primary role of automated tools, and somehow also provoke requirements and software engineers to find their position in the overall picture(s). However, with LLMs taking over creative aspects of software development, the role of humans risks being reduced to mere supervision, depriving them of the satisfaction and intellectual engagement that typically comes from the hands-on process of software creation. In the past, automation in software engineering was oriented to address tedious and error prone tasks, such as requirements review or tracing. This was expected to leave more space for requirements elicitation, modelling, and coding, which are arguably more exciting and less bureaucratic tasks. In a context such as the one envisioned, in which LLMs take all the interesting jobs, one could question the role of requirements engineers. We argue that requirements engineers, which are often neglected in software development [172], will actually become central. As software development becomes increasingly automated with the use of LLMs, requirements will emerge as the primary means of interacting with these models, especially for coding tasks. Indeed, in processes like the one outlined in Section 5, the inputs are the requirements, and the output is the code. As a result, software engineers will need to transition into roles as requirements engineers, orchestrating the entire development process through the management and specification of requirements.

Limited Training on FM datasets As briefly touched in Section 4, one challenge to overcome is that LLMs have been mainly trained on NL data, and their knowledge of formal languages may be limited. This especially affects the generation of formal artefacts from conventional ones and vice versa. Some ideas to deal with this aspect have already been mentioned in the paper. One can rely on the similarity between some formal languages and programming languages, and the code synthesis ability of LLMs specialised on code [113,114]. Fine-tuning LLMs on FM data is also a possibility, as well as integrating the output of different LLMs trained or fine-tuned on different types of data (NL, code, FM). Finally, one research avenue is an interactive generation, in which the human guides the generation of formal or conventional artefacts through iteration of prompts, answers, and corrections. This can possibly be further automated with an agent-based paradigm that enables LLMs to access model checkers and compilers, to assess the syntax correctness of the generated artefacts.

Proliferation and Maintainability of Artefacts Combining LLMs and FMs is expected, on the one hand, to speed up the development process and, on the other hand, to foster system correctness. At the same time, these advantages also come with more artefacts to maintain and keep consistent, i.e., more code, but also more models, formulae, counterexamples, etc. LLM-based automatic trace link generation, as discussed in Section 4, can help cope with artefact evolution, but the proliferation of the trace link themselves can become hardly manageable. Novel techniques are thus required to better monitor the evolution and control the increased pace and complexity of development. We argue that the software visualisation community can help in this direction [173]. Software visualisation tools can facilitate real-time monitoring of artefact changes, enabling teams to quickly identify inconsistencies and assess the impact of modifications across the system. Furthermore, integrating visualisation with analytics can provide predictive insights, helping teams anticipate potential issues arising from the rapid evolution of artefacts.

Deployment, Scalability and Technological Evolution

Radically changing an established software development approach by introducing FMs and LLMs is not a quick procedure, and requires careful consideration and planning. Research is needed not only to explore novel LLM-based and FM-controlled software processes but also to define effective strategies for *deploying* these processes in real-world software development contexts. To facilitate this transition, incremental process transformation approaches should be implemented, allowing for smooth integration and fostering acceptance among development teams.

Another significant challenge to address is the *scalability* of the proposed ideas. FMs, particularly model checkers, often face the state space explosion problem, which can complicate the verification of large systems. On the other hand, LLMs require substantial computational resources for execution, which can be a barrier to their widespread adoption. Solutions such as statistical model checking [174] or more recent proposals such as quantum probabilistic model checking [175], besides other techniques, can help mitigate the state space explosion issue. Furthermore, model distillation approaches [28] can reduce the resource demands associated with LLMs.

Lastly, the pace of *technological evolution* presents a unique challenge. The world of FMs traditionally develops industry-ready tools slowly, allowing companies ample time to adopt them. In contrast, new LLMs are emerging daily, which can overwhelm teams trying to keep up with the rapid advancements. Therefore, flexible approaches that can accommodate ongoing technological changes are essential. By addressing these challenges, the integration of FMs and LLMs into software development can be achieved in a manner that is both practical and sustainable.

8. Conclusion

We outlined two roadmaps: one aimed at ensuring the reliable use of LLMs in RE activities, and the other focused on enhancing the usability of formal RE through LLM support. On the one hand, our vision reimagines the application of formal RE by leveraging LLMs to make formal languages and tools more accessible, and on the other, the proposed use of formal techniques when LLMs are used in RE activities could mitigate concerns related to LLM-generated artefacts and establish a framework for responsible and trustworthy AI. The roadmaps are well-suited for rigorous software engineering processes, such as the V-process for mission-critical systems, where formal methods can play a key role in ensuring reliability and trustworthiness. However, they are not restricted to such approaches, as we believe that greater formality can provide benefits in any development context. This observation is especially important in a world in which safety, security, privacy, and ethical requirements are becoming increasingly relevant across many types of applications.

The primary aim of the roadmaps is to stimulate further research and reflection, while the authors will pursue part of the outlined directions. In the context of the roadmaps, our main focus will be on: (i) translating requirements into formal logic using LLMs, building on prior research initiated by the authors before the introduction of LLMs [131]; (ii) exploring the generation and analysis of software artefacts with LLMs, a line of work already being developed by one of the authors [9]; and (iii) leveraging LLMs to explain formal artefacts, which we identify as a critical challenge for the broader adoption of FMs. We hope that our roadmaps will inspire our readers to take over the other research avenues, and possibly develop new, unforeseen ones.

CRediT authorship contribution statement

Alessio Ferrari: Writing – review & editing, Writing – original draft, Methodology, Investigation, Formal analysis, Conceptualization. **Paola Spoletini:** Writing – review & editing, Writing – original draft, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This study was carried out within the MOST – Sustainable Mobility National Research Center and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1033 17/06/2022, CN00000023). This work was also partially supported by National MIUR-PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems), by the European Union's Horizon Europe research and innovation programme under grant agreement no. 101060179 (CODECS), and by the National Science Foundation, United States under grant CCF-1718377.

Data availability

No data was used for the research described in the article.

References

[1] W.X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al., A survey of large language models, 2023, arXiv preprint arXiv:2303.18223.

[2] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, J.M. Zhang, Large language models for software engineering: Survey and open problems, 2023, arXiv preprint arXiv:2310.03533.

[3] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large language models for software engineering: A systematic literature review, 2023, arXiv preprint arXiv:2308.10620.

[4] C. Jain, P.R. Anish, A. Singh, S. Ghaisas, A transformer-based approach for abstractive summarization of requirements from obligations in software engineering contracts, in: 2023 IEEE 31st International Requirements Engineering Conference, RE, IEEE, 2023, pp. 169–179.

[5] A. Marczak-Czajka, J. Cleland-Huang, Using ChatGPT to generate human-value user stories as inspirational triggers, in: 2023 IEEE 31st International Requirements Engineering Conference Workshops, REW, IEEE, 2023, pp. 52–61.

[6] A.D. Rodriguez, K.R. Dearstyne, J. Cleland-Huang, Prompts matter: Insights and strategies for prompt engineering in automated software traceability, in: 2023 IEEE 31st International Requirements Engineering Conference Workshops, REW, IEEE, 2023, pp. 455–464.

[7] B. Chen, K. Chen, S. Hassani, Y. Yang, D. Amyot, L. Lessard, G. Mussbacher, M. Sabetzadeh, D. Varró, On the use of GPT-4 for creating goal models: an exploratory study, in: 2023 IEEE 31st International Requirements Engineering Conference Workshops, REW, IEEE, 2023, pp. 262–271.

[8] J. Cámara, J. Troya, L. Burgueño, A. Vallecillo, On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML, Softw. Syst. Model. (2023) 1–13.

[9] A. Ferrari, S. Abualhaija, C. Arora, Model generation with LLMs: from requirements to UML sequence diagrams, in: 2024 IEEE 32st International Requirements Engineering Conference Workshops, REW, 2024, pp. 291–300.

[10] J.-M. Bruel, S. Ebersold, F. Galinier, M. Mazzara, A. Naumchev, B. Meyer, The role of formalism in system requirements, ACM Comput. Surv. 54 (5) (2021) 1–36.

[11] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, Cambridge, MA, USA, 1999.

[12] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, 1977, pp. 238–252.

[13] R.W. Floyd, Assigning meanings to programs, Proc. Symp. Appl. Math. 19 (1967) 19–32, http://dx.doi.org/10.1090/psapm/019/0224860.

[14] P. Spoletini, A. Ferrari, The return of formal requirements engineering in the era of large language models, in: D. Méndez, A. Moreira (Eds.), Requirements Engineering: Foundation for Software Quality - 30th International Working Conference, REFSQ 2024, Winterthur, Switzerland, April 8-11, 2024, Proceedings, in: Lecture Notes in Computer Science, vol. 14588, Springer, 2024, pp. 344–353, http://dx.doi.org/10.1007/978-3-031-57327-9_22.

[15] C.D. Manning, Introduction to Information Retrieval, Syngress Publishing, United Kingdom, 2008.

[16] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, Adv. Neural Inf. Process. Syst. 26 (2013).

[17] J. Firth, A synopsis of linguistic theory, 1930–1955, Stud. Linguist. Anal. (1957) 10–32.

[18] S.J. Johnson, M.R. Murty, I. Navakanth, A detailed review on word embedding techniques with emphasis on word2vec, Multimedia Tools Appl. 83 (13) (2024) 37979–38007.

[19] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, 2018, arXiv preprint arXiv:1810.04805.

[20] K.S. Kalyan, A. Rajasekharan, S. Sangeetha, Ammus: A survey of transformer-based pretrained models in natural language processing, 2021, arXiv preprint arXiv:2108.05542.

[21] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, J. Gao, Large language models: A survey, 2024, arXiv preprint arXiv:2402.06196.

[22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q.V. Le, D. Zhou, et al., Chain-of-thought prompting elicits reasoning in large language models, Adv. Neural Inf. Process. Syst. 35 (2022) 24824–24837.

[23] J. Liu, A. Liu, X. Lu, S. Welleck, P. West, R.L. Bras, Y. Choi, H. Hajishirzi, Generated knowledge prompting for commonsense reasoning, 2021, arXiv preprint arXiv:2110.08387.

[24] S. Qiao, Y. Ou, N. Zhang, X. Chen, Y. Yao, S. Deng, C. Tan, F. Huang, H. Chen, Reasoning with language model prompting: A survey, in: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Toronto, Canada, 2023, pp. 5368–5393.

[25] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu, et al., Instruction tuning for large language models: A survey, 2023, arXiv preprint arXiv:2308.10792.

[26] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al., Retrieval-augmented generation for knowledge-intensive NLP tasks, Adv. Neural Inf. Process. Syst. 33 (2020) 9459–9474.

[27] E.J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, Lora: Low-rank adaptation of large language models, 2021, arXiv preprint arXiv:2106.09685.

[28] X. Xu, M. Li, C. Tao, T. Shen, R. Cheng, J. Li, C. Xu, D. Tao, T. Zhou, A survey on knowledge distillation of large language models, 2024, arXiv preprint arXiv:2402.13116.

[29] J. Wei, M. Bosma, V.Y. Zhao, K. Guu, A.W. Yu, B. Lester, N. Du, A.M. Dai, Q.V. Le, Finetuned language models are zero-shot learners, 2021, arXiv preprint arXiv:2109.01652.
[30] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou, et al., The rise and potential of large language model based agents: A survey, 2023, arXiv preprint arXiv:2309.07864.
[31] L. Zhao, W. Alhoshan, A. Ferrari, K.J. Letsholo, M.A. Ajagbe, E.-V. Chioasca, R.T. Batista-Navarro, Natural language processing for requirements engineering: A systematic mapping study, ACM Comput. Surv. 54 (3) (2021) 1–41.
[32] A. Ferrari, G. Gori, B. Rosadini, I. Trotta, S. Bacherini, A. Fantechi, S. Gnesi, Detecting requirements defects with NLP patterns: an industrial experience in the railway domain, Empir. Softw. Eng. 23 (6) (2018) 3684–3733.
[33] J. Guo, J. Cheng, J. Cleland-Huang, Semantically enhanced software traceability using deep learning techniques, in: 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE, IEEE, 2017, pp. 3–14.
[34] W. Alhoshan, A. Ferrari, L. Zhao, Zero-shot learning for requirements classification: An exploratory study, Inf. Softw. Technol. 159 (2023) 107202.
[35] W. Maalej, Z. Kurtanović, H. Nabil, C. Stanik, On the automatic classification of app reviews, Requir. Eng. 21 (2016) 311–331.
[36] L. Montgomery, C. Lüders, W. Maalej, Mining issue trackers: Concepts and techniques, 2024, arXiv preprint arXiv:2403.05716.
[37] T. Hey, J. Keim, A. Koziolek, W.F. Tichy, Norbert: Transfer learning for requirements classification, in: 2020 IEEE 28th International Requirements Engineering Conference, RE, IEEE, 2020, pp. 169–179.
[38] S. Bashir, M. Abbas, A. Ferrari, M. Saadatmand, P. Lindberg, Requirements classification for smart allocation: A case study in the railway industry, in: 2023 IEEE 31st International Requirements Engineering Conference, RE, IEEE, 2023, pp. 201–211.
[39] M. Abbas, A. Ferrari, A. Shatnawi, E. Enoiu, M. Saadatmand, D. Sundmark, On the relationship between similar requirements and similar software: A case study in the railway domain, Requir. Eng. 28 (1) (2023) 23–47.
[40] Q. Motger, A. Miaschi, F. Dell'Orletta, X. Franch, J. Marco, T-FREX: A transformer-based feature extraction method from mobile app reviews, 2024, arXiv preprint arXiv:2401.03833.
[41] D. Luitel, S. Hassani, M. Sabetzadeh, Improving requirements completeness: Automated assistance through large language models, Requir. Eng. (2024) 1–23.
[42] K. Chen, Y. Yang, B. Chen, J.A.H. López, G. Mussbacher, D. Varró, Automated domain modeling with large language models: A comparative study, in: 2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems, MODELS, IEEE, 2023, pp. 162–172.
[43] X. Lian, J. Ma, H. Lv, L. Zhang, ReqCompletion: Domain-enhanced automatic completion for software requirements, in: 32nd IEEE International Requirements Engineering Conference, RE 2024, IEEE, 2024, pp. 142–154.
[44] S. Santos, T.D. Breaux, T.B. Norton, S. Haghighi, S. Ghanavati, Requirements satisfiability with in-context learning, in: 32nd IEEE International Requirements Engineering Conference, RE 2024, IEEE, 2024, URL: https://arxiv.org/abs/2404.12576.
[45] B. Görer, F.B. Aydemir, GPT-powered elicitation interview script generator for requirements engineering training, in: 32nd IEEE International Requirements Engineering Conference, RE@Next! 2024, IEEE, 2024, pp. 372–379.
[46] M. Krishna, B. Gaur, A. Verma, P. Jalote, Using LLMs in software requirements specifications: An empirical evaluation, in: 32nd IEEE International Requirements Engineering Conference, RE@Next! 2024, IEEE, 2024, pp. 475–483.
[47] S. Lubos, A. Felfernig, T.N.T. Tran, D. Garber, M.E. Mansi, S.P. Erdeniz, V.M. Le, Leveraging LLMs for the quality assurance of software requirements, in: 32nd IEEE International Requirements Engineering Conference, RE@Next! 2024, IEEE, 2024, pp. 389–397.
[48] B. Wei, Requirements are all you need: From requirements to code with LLMs, in: 32nd IEEE International Requirements Engineering Conference, RE@Next! 2024, IEEE, 2024, pp. 416–422.
[49] N. Feng, L. Marsso, S.G. Yaman, I. Standen, Y. Baatartogtokh, R. Ayad, R. Calinescu, M. Chechik, Normative requirements operationalization with large language models, in: 32nd IEEE International Requirements Engineering Conference, RE 2024, IEEE, 2024, pp. 129–141.
[50] J. Spivey, The Z Notation: A Reference Manual, Prentice Hall, 1992.
[51] J.R. Büchi, On a decision method in restricted second-order arithmetic, Int. Congr. Log. Methodol. Philos. Sci. (1962) 1–11.
[52] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, sfcs 1977, 1977, pp. 46–57.
[53] President's Commission on the Accident at Three Mile Island, Report of the president's commission on the accident at three mile island: The need for change: The legacy of TMI, 1979, https://www.hsdl.org/?abstract&did=3828.
[54] J.P. Bowen, M.G. Hinchey (Eds.), Using Z: Specification, Refinement, and Proof, Prentice Hall, 1996.
[55] J.M. Spivey, Understanding Z: A Specification Language and its Formal Semantics, Cambridge University Press, 1988.
[56] C. Jones, Systematic Software Development using VDM, Prentice Hall, 1990.
[57] D. Bjørner, C.B. Jones, The Vienna Development Method: The Meta-Language, Springer-Verlag, 1982.
[58] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, M. Verhoef, Validated Designs for Object-oriented Systems, Springer, 2005.
[59] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, M. Verhoef, Validated Designs for Object-oriented Systems: VDMTools, Springer, 2005, VDMTools is discussed extensively as part of the practical applications of VDM in this book.
[60] P.G. Larsen, N. Battle, S. Brydon, K. Pierce, P. Troubitsyna, The overture initiative - integrating tools for VDM, in: FMCO 2009: Formal Methods for Components and Objects, Springer, 2010, pp. 54–72, http://dx.doi.org/10.1007/978-3-642-25271-6_4.
[61] J.R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, Cambridge, UK, 1996.
[62] R. Milner, A Calculus of Communicating Systems, Springer, 1980.
[63] C. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
[64] ITU-T, Specification and Description Language (SDL), 1988.
[65] CoFI (The Common Framework Initiative), CASL Reference Manual, Springer, 2004.
[66] ISO, LOTOS - A formal description technique based on the temporal ordering of observational behaviour, 1989, ISO 8807.
[67] L. Lamport, The temporal logic of actions, ACM Trans. Program. Lang. Syst. (TOPLAS) 16 (3) (1994) 872–923, http://dx.doi.org/10.1145/177492.177726.
[68] D. Jackson, Software Abstractions: Logic, Language, and Analysis, MIT Press, 2012.
[69] Fizzbee, A Python-inspired design specification language, https://fizzbee.io/.
[70] N.E. Fuchs, U. Schwertel, R. Schwitter, Attempto Controlled English (ACE), in: Proceedings of the Workshop on Controlled Natural Language, CNL, 1996.
[71] M. Huth, M. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press, 2004.
[72] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, FOCS, 1977, pp. 46–57, http://dx.doi.org/10.1109/SFCS.1977.32.
[73] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. (TOPLAS) 8 (2) (1986) 244–263, http://dx.doi.org/10.1145/5397.5399.
[74] E.A. Emerson, J.Y. Halpern, "Sometimes" and "not never" revisited: On branching versus linear time temporal logic, J. ACM 33 (1) (1986) 151–178, http://dx.doi.org/10.1145/49051.49054.
[75] D. Kozen, Results on the propositional μ-calculus, Theoret. Comput. Sci. 27 (1983) 333–354, http://dx.doi.org/10.1016/0304-3975(82)90125-6.
[76] M.J.C. Gordon, T.F. Melham, Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, Cambridge, UK, 1993.
[77] P. Blackburn, M. de Rijke, Y. Venema, Modal Logic, Cambridge University Press, 2001.
[78] R. Alur, T.A. Henzinger, A really temporal logic, J. ACM 41 (1) (1994) 181–204, http://dx.doi.org/10.1145/174644.174651.
[79] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.
[80] M.Y. Vardi, P. Wolper, Reasoning about Infinite Computations, Springer, 1994.
[81] J.-P. Katoen, Probabilistic model checking, in: Proceedings of the 12th International Conference on Computer Aided Verification, CAV, 2001, pp. 358–372, http://dx.doi.org/10.1007/3-540-44585-4_34.
[82] H. Hermanns, U. Herzog, J.-P. Katoen, Process algebra for performance evaluation, Theoret. Comput. Sci. 274 (1–2) (2002) 43–87.
[83] D. Harel, Statecharts: A visual formalism for complex systems, Sci. Comput. Program. 8 (3) (1987) 231–274, http://dx.doi.org/10.1016/0167-6423(87)90035-9.
[84] R. Alur, D.L. Dill, Hierarchical state machines, in: Proceedings of the 28th Annual ACM Symposium on Theory of Computing, STOC, 1996, pp. 671–682, http://dx.doi.org/10.1145/237814.237983.
[85] P. Fritzson, Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica, Wiley-IEEE Press, 2011.
[86] C. Ptolemaeus, System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2003.
[87] T. Murata, Petri Nets: Properties, analysis and applications, Proc. IEEE 77 (4) (1989) 541–580, http://dx.doi.org/10.1109/5.24143.
[88] K. Jensen, Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, vol. 1, Springer, 1997.
[89] C.V. Ramchandani, Analysis of Asynchronous Concurrent Systems by Timed Petri Nets, Project MAC Technical Report, 1974.
[90] M.K. Molloy, Performance analysis using stochastic Petri nets, IEEE Trans. Comput. 31 (9) (1982) 913–917, http://dx.doi.org/10.1109/TC.1982.1676110.
[91] H.J. Genrich, K. Lautenbach, System modelling with high-level Petri nets, in: Theoretical Computer Science, vol. 13, 1981, pp. 109–135, http://dx.doi.org/10.1016/0304-3975(81)90100-4.
[92] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977, pp. 238–252, http://dx.doi.org/10.1145/512950.512973.

[93] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, A static analyzer for large safety-critical software, in: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, 2003, pp. 196–207, http://dx.doi.org/10.1145/781131.781153.
[94] Facebook, Infer static analyzer, 2015, https://fbinfer.com/.
[95] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer, 1999.
[96] Coverity, Coverity static analysis, 2010, https://www.coverity.com/.
[97] PVS-Studio, PVS-studio static analysis tool, 2010, https://www.viva64.com/en/pvs-studio/.
[98] FindBugs, FindBugs static analysis tool, 2007, http://findbugs.sourceforge.net/.
[99] G.J. Holzmann, The model checker SPIN, IEEE Trans. Softw. Eng. 23 (5) (1997) 279–295, http://dx.doi.org/10.1109/32.588521.
[100] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14, Springer, 2002, pp. 359–364, http://dx.doi.org/10.1007/3-540-45657-0_29.
[101] G. Behrmann, A. David, K.G. Larsen, A tutorial on uppaal, Form. Methods Real-Time Syst. (2004) 200–236, http://dx.doi.org/10.1007/978-3-540-30080-9_7.
[102] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, Springer, 2004.
[103] o. TNipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL: A proof assistant for higher-order logic, in: Lecture Notes in Computer Science, vol. 2283, Springer, 2002.
[104] U. Norell, Dependently typed programming in agda, in: Proceedings of the 4th International Workshop on Types in Language Design and Implementation, TLDI'09, 2009, pp. 1–2, http://dx.doi.org/10.1145/1481861.1481862.
[105] P. Cuoq, J. Signoles, N. Kosmatov, V. Prevosto, B.-Y.E. Chang, X. Rival, Frama-C: A oftware analysis perspective. proceedings of the 10th international conference on software engineering and formal methods, SEFM 2012, 2012, pp. 233–247, http://dx.doi.org/10.1007/978-3-642-33826-7_16.
[106] K.R.M. Leino, Dafny: An automatic program verifier for functional correctness, LPAR 2010, in: Proceedings of the 16th International Con- ference on Logic for Programming, Artificial Intelligence, and Rea- soning, 2010, pp. 348–370, http://dx.doi.org/10.1007/978-3-642-17511-4_20.
[107] B. Beckert, R. Hähnle, P.H. Schmitt, Verification of Object-Oriented Software: The KeY Approach, Springer, 2007.
[108] N. Wirth, Program development by stepwise refinement, Commun. ACM 14 (4) (1971) 221–227, http://dx.doi.org/10.1145/362575.362577.
[109] J.R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, 2010.
[110] G.J. Holzmann, Promela Manual Pages, 2003, URL: http://spinroot.com/spin/Man/promela.html.
[111] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, Boston, 2003.
[112] M. Eilers, P. Müller, Nagini: a static verifier for Python, in: Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30, Springer, 2018, pp. 596–603.
[113] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X.E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al., Code Llama: Open foundation models for code, 2023, arXiv preprint arXiv:2308.12950.
[114] R. Li, L.B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al., Starcoder: may the source be with you!, 2023, arXiv preprint arXiv:2305.06161.
[115] H. Koziolek, S. Grüner, R. Hark, V. Ashiwal, S. Linsbauer, N. Eskandani, LLM-based and retrieval-augmented control code generation, in: Proc. 1st Int. Workshop on Large Language Models for Coffice (LLM4Code) at ICSE, vol. 2024, 2024.
[116] T. Zhu, Z. Li, M. Pan, C. Shi, T. Zhang, Y. Pei, X. Li, Deep is better? An empirical comparison of information retrieval and deep learning approaches to code summarization, ACM Trans. Softw. Eng. Methodol. 33 (3) (2024) 1–37.
[117] R. Jhala, R. Majumdar, Software model checking, ACM Comput. Surv. 41 (4) (2009) 1–54.
[118] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, C. Trippel, nl2spec: Interactively translating unstructured natural language to temporal logics with large language models, in: International Conference on Computer Aided Verification, Springer, 2023, pp. 383–396.
[119] R. Zrelli, H. Amaral Misson, M. Ben Attia, F. Gohring de Magalhães, A. Shabah, G. Nicolescu, Natural2CTL: A dataset for natural language requirements and their CTL formal equivalents, in: International Working Conference on Requirements Engineering: Foundation for Software Quality, Springer, 2024, pp. 205–216.
[120] E.A. Emerson, A.K. Mok, A.P. Sistla, J. Srinivasan, Quantitative temporal reasoning, Real-Time Syst. 4 (4) (1992) 331–352.
[121] E.A. Emerson, Model checking and the mu-calculus., Descr. Complex. Finite Model. 31 (1996) 185–214.
[122] A. Hartmanns, M. Klauck, D. Parker, T. Quatmann, E. Ruijters, The quantitative verification benchmark set, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2019, pp. 344–350.
[123] M. Jasper, M. Mues, M. Schlüter, B. Steffen, F. Howar, Rers 2018: Ctl, ltl, and reachability, in: Leveraging Applications of Formal Methods, Verification and Validation. Verification: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II 8, Springer, 2018, pp. 433–447.
[124] F. Mazzanti, A. Ferrari, Ten diverse formal models for a CBTC automatic train supervision system, 2018, arXiv preprint arXiv:1803.10324.
[125] E. Kamburjan, R. Hähnle, Prototyping formal system models with active objects, in: 11th Interaction and Concurrency Experience, 2018.
[126] M. Gleirscher, J. van de Pol, J. Woodcock, A manifesto for applicable formal methods, Softw. Syst. Model. 22 (6) (2023) 1737–1749.
[127] A. Ferrari, F. Mazzanti, D. Basile, M.H.t. Beek, A. Fantechi, Comparing formal tools for system design: a judgment study, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 62–74.
[128] J. Barnat, L. Brim, J. Beran, Í.R. Oliveira, et al., Executing model checking counterexamples in simulink, in: 2012 Sixth International Symposium on Theoretical Aspects of Software Engineering, IEEE, 2012, pp. 245–248.
[129] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, X. Liao, Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.
[130] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, B. Myers, Using an LLM to help with code understanding, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
[131] H. Cherukuri, A. Ferrari, P. Spoletini, Towards explainable formal methods: From LTL to natural language with neural machine translation, in: REFSQ, Springer, 2022, pp. 79–86.
[132] M. Balfroid, B. Vanderose, X. Devroey, Towards LLM-generated code tours for onboarding, in: Workshop on NL-Based Software Engineering, NLBSE'24, 2024.
[133] D. Basile, M.H. ter Beek, A. Ferrari, A. Legay, Modelling and analysing ERTMS L3 moving block railway signalling with simulink and uppaal SMC, in: Formal Methods for Industrial Critical Systems: 24th International Conference, FMICS 2019, Amsterdam, the Netherlands, August 30–31, 2019, Proceedings 24, Springer, 2019, pp. 1–21.
[134] A. Ferrari, F. Mazzanti, D. Basile, M.H. ter Beek, Systematic evaluation and usability analysis of formal methods tools for railway signaling system design, IEEE Trans. Softw. Eng. 48 (11) (2021) 4675–4691.
[135] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Schumann, Automated formalization of structured natural language requirements, Inf. Softw. Technol. 137 (2021) 106590, http://dx.doi.org/10.1016/j.infsof.2021.106590.
[136] F. Mazzanti, A. Ferrari, G.O. Spagnolo, Towards formal methods diversity in railways: an experience report with seven frameworks, Int. J. Softw. Tools Technol. Transf. 20 (2018) 263–288.
[137] R. Pan, A.R. Ibrahimzada, R. Krishna, D. Sankar, L.P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, R. Jabbarvand, 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, 2024, pp. 1–13.
[138] S. Anand, E.K. Burke, T.Y. Chen, J. Clark, M.B. Cohen, W. Grieskamp, M. Harman, M.J. Harrold, P. McMinn, A. Bertolino, et al., An orchestrated survey of methodologies for automated software test case generation, J. Syst. Softw. 86 (8) (2013) 1978–2001.
[139] Z. Pauzi, A. Capiluppi, Applications of natural language processing in software traceability: A systematic mapping study, J. Syst. Softw. 198 (2023) 111616.
[140] B.P. Allen, L. Stork, P.T. Groth, Knowledge engineering using large language models, 2023, ArXiv abs/2310.00637.
[141] P. Mateiu, A. Groza, Ontology engineering with large language models, in: 2023 25th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC, IEEE, 2023, pp. 226–229.
[142] R. Buchmann, J. Eder, H.-G. Fill, U. Frank, D. Karagiannis, E. Laurenzi, J. Mylopoulis, D. Plexousakis, M.Y. Santos, Large language models: Expectations for semantics-driven systems engineering, 2024, https://ssrn.com/abstract=4633719 or http://dx.doi.org/10.2139/ssrn.4633719.
[143] H. Yang, A. De Roeck, V. Gervasi, A. Willis, B. Nuseibeh, Analysing anaphoric ambiguity in natural language requirements, Requir. Eng. 16 (2011) 163–189.
[144] T. Ahmed, P. Devanbu, Few-shot training LLMs for project-specific code-summarization, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–5.
[145] X. Zhou, K. Kim, B. Xu, D. Han, J. He, D. Lo, Generation-based code review automation: how far are we? in: 2023 IEEE/ACM 31st International Conference on Program Comprehension, ICPC, IEEE, 2023, pp. 215–226.
[146] Y. Elrakaiby, A. Ferrari, P. Spoletini, S. Gnesi, B. Nuseibeh, Using argumentation to explain ambiguity in requirements elicitation interviews, in: 2017 IEEE 25th International Requirements Engineering Conference, RE, IEEE, 2017, pp. 51–60.

[147] Y. Elrakaiby, A. Ferrari, J. Mylopoulos, Care: A refinement calculus for requirements engineering based on argumentation semantics, in: 2018 IEEE 26th International Requirements Engineering Conference, RE, IEEE, 2018, pp. 364–369.

[148] I. Yildirim, L. Paul, From task structures to world models: what do LLMs know? Trends Cogn. Sci. (2024).

[149] L. Sarker, M. Downing, A. Desai, T. Bultan, Syntactic robustness for LLM-based code generation, 2024, arXiv preprint arXiv:2404.01535.

[150] Z. Azerbayev, H. Schoelkopf, K. Paster, M.D. Santos, S. McAleer, A.Q. Jiang, J. Deng, S. Biderman, S. Welleck, Llemma: An open language model for mathematics, 2023, arXiv preprint arXiv:2310.10631.

[151] I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, M. Farajtabar, GSM-symbolic: Understanding the limitations of mathematical reasoning in large language models, 2024, arXiv preprint arXiv:2410.05229.

[152] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R.J. Prenger, A. Anandkumar, Leandojo: Theorem proving with retrieval-augmented language models, Adv. Neural Inf. Process. Syst. 36 (2024).

[153] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, G. Neubig, Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, ACM Comput. Surv. 55 (9) (2023) 1–35.

[154] S. Lee, W. Sim, D. Shin, S. Hwang, W. Seo, J. Park, S. Lee, S. Kim, S. Kim, Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus, 2024, arXiv preprint arXiv:2403.11793.

[155] C.R. Gallistel, A.P. King, Memory and the Computational Brain: Why Cognitive Science Will Transform Neuroscience, John Wiley & Sons, 2011.

[156] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F.L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al., Gpt-4 technical report, 2023, arXiv preprint arXiv:2303.08774.

[157] Z.Z. Chen, J. Ma, X. Zhang, N. Hao, A. Yan, A. Nourbakhsh, X. Yang, J. McAuley, L. Petzold, W.Y. Wang, A survey on large language models for critical societal domains: Finance, healthcare, and law, 2024, arXiv preprint arXiv:2405.01769.

[158] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, D. Zhou, Self-consistency improves chain of thought reasoning in language models, 2022, arXiv preprint arXiv:2203.11171.

[159] G. Singh, T. Gehr, M. Mirman, M. Püschel, M. Vechev, Fast and effective robustness certification, in: S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), in: Advances in Neural Information Processing Systems, vol. 31, Curran Associates, Inc., 2018.

[160] F. Boudardara, A. Boussif, P.-J. Meyer, M. Ghazel, A review of abstraction methods towards verifying neural networks, ACM Trans. Embed. Comput. Syst. (2023).

[161] O. Amaral, M.I. Azeem, S. Abualhaija, L.C. Briand, Nlp-based automated compliance checking of data processing agreements against GDPR, 2023, IEEE TSE.

[162] T.D. Breaux, D.G. Gordon, Regulatory requirements traceability and analysis using semi-formal specifications, in: REFSQ, Springer, 2013, pp. 141–157.

[163] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to runtime verification, Lect. Runtime Verification: Introd. Adv. Top. (2018) 1–33.

[164] B. Wang, W. Chen, H. Pei, C. Xie, M. Kang, C. Zhang, C. Xu, Z. Xiong, R. Dutta, R. Schaeffer, S.T. Truong, S. Arora, M. Mazeika, D. Hendrycks, Z. Lin, Y. Cheng, S. Koyejo, D. Song, B. Li, DecodingTrust: A comprehensive assessment of trustworthiness in GPT models, in: A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, S. Levine (Eds.), Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, la, USA, December 10 - 16, 2023, 2023, URL: http://papers.nips.cc/paper_files/paper/2023/hash/63cb9921eecf51bfad27a99b2c53dd6d-Abstract-Datasets_and_Benchmarks.html.

[165] D. Huang, Q. Bu, J. Zhang, X. Xie, J. Chen, H. Cui, Bias testing and mitigation in LLM-based code generation, 2024, arXiv:2309.14345.

[166] F.B. Aydemir, F. Dalpiaz, A roadmap for ethics-aware software engineering, in: Proceedings of the International Workshop on Software Fairness, FairWare '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 15–21, http://dx.doi.org/10.1145/3194770.3194778.

[167] X. Franch, A. Jedlitschka, S. Martínez-Fernández, A requirements engineering perspective to AI-based systems development: A vision paper, in: A. Ferrari, B. Penzenstadler (Eds.), Requirements Engineering: Foundation for Software Quality - 29th International Working Conference, REFSQ 2023, Barcelona, Spain, April 17-20, 2023, Proceedings, in: Lecture Notes in Computer Science, vol. 13975, Springer, 2023, pp. 223–232, http://dx.doi.org/10.1007/978-3-031-29786-1_15.

[168] M.H. ter Beek, A. Ferrari, Empirical formal methods: guidelines for performing empirical studies on formal methods, Software 1 (4) (2022).

[169] V. Braun, V. Clarke, Thematic Analysis, American Psychological Association, 2012.

[170] R. Hoda, Qualitative Research with Socio-Technical Grounded Theory, Springer, 2024.

[171] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen, et al., Siren's song in the AI ocean: a survey on hallucination in large language models, 2023, arXiv preprint arXiv:2309.01219.

[172] A. Herrmann, Requirements engineering in practice: There is no requirements engineer position, in: Requirements Engineering: Foundation for Software Quality: 19th International Working Conference, REFSQ 2013, Essen, Germany, April 8-11, 2013. Proceedings 19, Springer, 2013, pp. 347–361.

[173] N. Chotisarn, L. Merino, X. Zheng, S. Lonapalawong, T. Zhang, M. Xu, W. Chen, A systematic literature review of modern software visualization, J. Vis. 23 (2020) 539–558.

[174] A. Legay, B. Delahaye, S. Bensalem, Statistical model checking: An overview, in: International Conference on Runtime Verification, Springer, 2010, pp. 122–135.

[175] S. Jeon, K. Cho, C.G. Kang, J. Lee, H. Oh, J. Kang, Quantum probabilistic model checking for time-bounded properties, Proc. ACM Program. Lang. 8 (OOPSLA2) (2024) 557–587.