RQ2: What are the emerging trends and future research directions in using LLMs for software requirements formalisation?

The structure of the paper is organized as follows. Section 2 presents the methodology used to select the relevant literature. Section 3 provides a brief overview of works focused on the formalisation of software requirements using large language models (LLMs) addressing **RQ1**. Literature concerning the traceability of software requirements is discussed in Section 4. Section 5 lists down well-known formal notations developed over the last three decades of research by the formal method community and its related tools. Section 6 explores studies involving formal proofs within the frameworks of Unifying Theories of Programming (UTP) and the Theory of Institutions. Section 7 presents future directions based on our findings and the discussion on chain of thought and prompt engineering in sub-section 7.1, addressing **RQ2**. Concluding remarks are provided in Section 8.

2 METHODOLOGY FOR LITERATURE REVIEW

To conduct a structured and thorough review of literature on Natural Language Processing (NLP), Large Language Models (LLMs), and their use in software requirements, the following approach is followed. Several academic databases, including IEEE Xplore, ACM Digital Library, Scopus, Springer Link, and Google Scholar, are searched using specific keywords. The core search terms include "NLP," "LLMs," and "Software Requirements," with broader terms such as "specification," "logic," "verification," "model checking," and "theorem proving" used to expand the scope. The number of results differs notably across databases. For example, IEEE Xplore returns 17 peer-reviewed articles, Scopus lists 20, Springer Link filters to 595, ACM Digital Library provides 1,368 results, and Google Scholar shows 14,800 references since 2021. These discrepancies highlight the importance of applying precise selection methods to extract the most relevant studies.

To streamline the process of locating strong contributions, the AI-powered tool Elicit [20] is used. Elicit supports the literature review by offering summarised content and DOIs for suggested papers. While it helps reduce the manual workload during the initial phase, every suggested paper in this review is manually reviewed to ensure its relevance. This ensures that the final list excludes any unrelated or off-topic material. After the initial filtering, a manual review is performed to confirm the relevance and quality of each paper. Abstracts are first assessed to judge suitability. If the abstract lacks clarity or depth, a further examination of the full text is conducted.

Abstracts are closely read, and when necessary, the full text is reviewed using the following exclusion and inclusion criteria:

Inclusion Criteria: Studies are included if they offer meaningful theoretical or empirical insights related to NLP, LLMs, and their application in software requirements. This includes topics like specification, formal logic, verification, and formal methods.

Exclusion Criteria: Papers are excluded if they show in-sufficient relevance to the intersection of NLP/LLMs and software requirements, or if their abstracts or full texts lack sufficient detail. Non-peer-reviewed materials, duplicates, and items suggested by Elicit but deemed irrelevant after manual review are also removed.

3 FORMALISING REQUIREMENTS THROUGH LLMS

The paper [18] proposes using LLMs, like GPT-3.5, to verify code by analysing requirements and explaining whether they are met. The work [17] details about nl2spec, a framework that leverages LLMs to generate formal specifications from natural language, addressing the challenge of ambiguity in system requirements. Users can iteratively refine

translations, making formalization easier. The work [17] provides an open-source implementation with a web-based interface.

The work [73] provides verification and refinement of natural language explanations by making LLMs and theorem provers work together. A neuro-symbolic framework i.e. Explanation-Refiner is represented. LLMs and theorem provers are integrated together to formalise explanatory sentences. The theorem prover then provides the guarantee of validated sentence explanations. Theorem prover also provides feedback for further improvements in NLI (Natural Language Inference) model. Error correction mechanisms can also be deployed by using the tool Explanation-Refiner. Consequently, it automatically enhances the quality of explanations of variable complexity. [4] outlines key research directions for the stages of software requirement engineering, conducts a SWOT analysis, and share findings from an initial evaluation.

In [50], symbolic NLP and ChatGPT performance is compared while generating correct JML output against given natural language pre-conditions. In [5], domain models are generated against natural language requirements for industry-based case study. Domain model extractor is designed and applied to four industrial requirements documents. The accuracy and overall performance is reported for the designed model extractor. An automatic synthesis of software specifications is provided through LLMs in [59]. We know that software configurations give an insight of how the software will behave. While software are frequently discussed and documented in a variety of external sources, including software manuals, code comments, and online discussion forums, making it hard for system administrators to get the most optimal configurations due to lack of clarity and organisation in provided documentation. Work [59] proposed SpecSyn framework that uses an advanced language model to automatically generate software specifications from natural language text. It treats specification generation as a sequence-to-sequence learning task and outperforms previous tools by 21% in accuracy, extracting specifications from both single and multiple sentences.

AssertLLM tool is presented in [23]. The tool generates assertions to do hardware verification from design specifications, exploiting three customised LLMs. It is done in three phases, first understanding specifications, mapping signal definitions and generating assertions. The results show that AssertLLM produced 89% correct assertions with accurate syntax and function. The work [30] reports on formal verification of NASA's Node Control Software natural language specifications. The software is deployed at International Space Station. Errors found in the natural language requirements are reported by the authors with a commentary on lessons learnt.

SpecLLM [54] explores the space of generating and reviewing VLSI design specifications with LLMs. The cumbersome task of chip architects can be improved by exploiting the power of LLMs for synthesising natural language specifications involved in chip designing. So, the utility of LLMs is explored with the two stages i.e. (1) **generation** of architecture specifications from scratch and from register transfer logic (RTL) code; and (2) **reviewing** these generated specifications.

The paper [68] introduced a model-based language (Requirements Specification Language - RSL) that enhances software requirements specification by incorporating constrained natural language phrases, including verbs, adjectives, and prepositions, grouped by nouns. It also presented an advanced tooling framework that captures application logic specifications, enabling automated transformations into code, validated through a controlled experiment. The framework functionality is integrated as development platform named ReDSeeDS. The work was a part of EU project, maintained at www.redseeds.eu [68]. A similar work is reported in [31] which describes the details of ARSENAL framework and methodology designed to perform automatic requirements specification extraction from natural language. The generated specification can be verified automatically through the framework.

An interesting work is presented in [51]. Business process models specifes the requirements of process-aware information systems. It includes generation of natural language from business process models. Domain experts and system analysts find it difficult to validate BPMs directly. The automation of generating a text which describes these

models is done in [51]. For the translation process, BPMs are translated to RPST - a tree like structure of the process model which is then used to generate sentences first. These generated sentences are refined by adding linguistic complexity later-on. The generated natural language is found complete and more understandable. In primitive work of 1996 [69], the software requirements were expressed in a limited set of natural language. Authors [69] referred it as controlled natural language. For Controlled Language (CL) , authors used the Alvey Natural Language Toolkit (ANLT). This intermediate notation is then translated to logical expressions in order to detect and remove ambiguities in requirement specifications. In [74], the potential and power of LLMs is exploited for smart grid requirement specifications improvement. Here, the performance of GPT-4o and Claude 3.5 Sonnet is analysed through f1-scores, achieving in range of 79% - 94%.

The paper [91] reports the translation between NL and Linear Temporal Logic (LTL) formulas through the use of LLMs. The challange of low accuracy and high cost during model training and tuning of general purpose LLMs is considered in [91]. Dynamic prompt generation and human interaction with LLMs are amalgamated to deal with the mentioned challenges. Unstructured natural language requirements are converted to NL-LTL pairs. The approach achieved up to 94.4% accuracy on publicly available datasets with 36 and 255,000 NL-LTL pairs. Dealing with a different domain domain, it improved from 27% to 78% through interactive prompt evolution.

In [81], authors present ESBMC-AI, a framework that combined Large Language Models (LLMs) with Formal Verification to automatically detect and fix software vulnerabilities. The approach used Bounded Model Checking (BMC) to identify errors and generate counterexamples, which were then fed into LLM to repair the code, followed by re-verification with BMC. Evaluated on 50,000 C programs from the FormAI dataset, ESBMC-AI effectively fixed issues like buffer overflows and pointer dereference failures with high accuracy, making it a valuable tool for software development and CI/CD integration. In [38], authors reported performance of LLMs to translate natural language into formal rules by training them on datasets for regular expressions (regex), first-order logic (FOL), and linear-time temporal logic (LTL) . The results show that the models adapt well to new terms and symbols, and outperformed existing methods in regular expression translation.

In [62], SynVer is presented. SynVer is a framework to synthesise and verify C programs. Built on the Verified Software Toolchain, the tool usage was applied on benchmarks containing basic coding tasks, Separation Logic assertions, and API specifications. In [72], five different benchmark datasets are used to analyse the performance of GPT-3.5 and GPT-4, making sure the low-level software requirements meet all high-level requirements. The results reported in [72] showed that GPT-3.5, using zero-shot prompting with explanations, correctly detected full coverage in four out of five datasets and achieved 99.7% recall in spotting missing coverage from a removed low-level requirement.

SAT-LLM, a unique framework to remove conflicting requirements is represented in [24]. It integrated Satisfiability Modulo Theories (SMT) solvers with LLMs. The performance of LLMs struggles while removing complex requirements conflicts. With the capability of formal reasoning, integrating SMT solvers with LLMs makes it a viable approach for the task. Experiments are performed with SAT-LLM and it performed well as compared to standalone performance of LLM i.e. ChatGPT identified 33% of conflicts with a Precision of 0.85, Recall of 0.31, and an F1 score of 0.46, struggling with hidden or complex conflicts. SAT-LLM performed better, identifying 80% of conflicts with a Precision of 1.00, Recall of 0.83, and an F1 score of 0.91. [21] used input of error messages, variable names, procedure documentation and user questions. The suitablility of NLP techniques is discussed by going through the literature for each step of software development life cycle. The authors of [21] indicated NLP a good candidate for software development. For instance, they discussed available literature for generating assertions by synthesising sentences in testing phase.

The paper [63] introduced Req2Spec, an NLP-based tool that analyses natural language requirements to create formal specifications for HANFOR, a large-scale requirements and test generation tool. Tested on 222 automotive software

requirements at BOSCH, it correctly formalized 71% of them. A primitive work in the domain is about RML [36]. RML bundled with features of writing requirements, which are based on conceptual model, having attribute of organisation and abstraction, maintaining precision, consistency and clarity. Well-defined logic and semantics of RML are presented in [36]. The work [22] evaluated GPT-4o's ability to generate specifications for C programs that can be verified using VeriFast, a static verifier based on separation logic. Their experiments, which use different user inputs and prompting techniques, show that while GPT-4o's specifications maintain functional behaviour, they often fail verification and include redundancies when verifiable. The primitive work [66] described automatic translation from natural language sentences to temporal logic, in order to deploy formal verification of the requirements.

Paper [89] introduced a methodology Lemur that integrated large language models with automated reasoners for program verification, formally defining transition rules, proving their soundness, and demonstrating practical improvements on synthetic and competition benchmarks. Performance of Lemur is compared with Code2Inv, ESBMC, and UAutomizer while using SV-COMP benchmarks. Code2Inv is a comprehensive work of 2020 [80], contributing towards learning-based end-to-end program verification. Code2Inv utilised reinforcement learning to train an invariant synthesizer to propose loop invariants. [65] is a systematic review of the domain i.e. translating between natural language software requirements to formal ones, based on the databases of Scopus, ACM, IEEE Xplore and Clarivate. [67] proposed a pipeline integrating Large Language Models (LLMs) to automatically refine and decompose safety requirements for autonomous driving, addressing frequent updates in the automotive domain. The work evaluated LLMs' ability to support Hazard Analysis and Risk Assessment (HARA) through iterative design science and expert assessments, ultimately implementing the prototype in an industrial setting. The responsible software development team evaluated its efficiency.

[92] presented a framework to ensure consistency between different representations of specifications, maintaining semantic alignment between oral and formal descriptions while ensuring implementation potential through synthesis. It included time extraction, input-output partitioning, and semantic reasoning beyond syntactic parsing. The framework of [92] performed well on various test examples. The work [64] reports available NLP tools for Topological Functioning Modelling (TFM). Among six selected LLM pipelines, best performance tools found were the Stanford CoreNLP toolkit, FreeLing, and NLTK toolkit. In [60], power of LLMs is utilised for generating Dafny tasks. 178 problems are selected from MBPP benchmark. Three types of prompts are used: (1) Context less prompts, (2) Signature prompt comprised of method signature and test cases, and (3) Context of Thought (CoT) prompt based on decomposition of problems into multiple steps and inclusion of retrieval augmentation generated problems and solutions. GPT-4 outperformed PaLM-2 on the evaluated tasks and achieved the best results using the retrieval-augmented CoT prompt. The technique described in [60] provided 153 verified Dafny solutions to MBPP problems, including 103 synthesized by GPT-4 and 50 written manually.

The work [93] introduced LeanDojo, an open-source toolkit that enables programmatic interaction with Lean theorem prover, providing annotated proof data to support premise selection in theorem proving. Using LeanDojo's extracted data, [93] developed ReProver, a retrieval-augmented LLM-based prover that effectively selects premises, significantly improving theorem proving efficiency while requiring only one GPU week of training. Furthermore, a new benchmark with 98,734 theorems and proofs from Lean's math library constructed. It is designed to test generalization to novel premises, and demonstrate ReProver's superiority over non-retrieval baselines and GPT-4. Thor [47] is a framework developed to integrate language models with theorem provers. The task of finding relevant premises while proving conjecture is the crucial task in implementing automatic theorem provers. Thor [47] is presented to handle this very task. The class methods for selecting relevant premises are named Hammers. To test the performance of the framework,

the PISA dataset is used. It improved the accuracy from 39% to 57%, while 8.2% of the proofs could not be solved neither from the language models nor from the theorem provers. Thor also outperformed the previous works while using MiniF2F dataset. The paper [35] explores the space of integrating Co-pilot (github) with formal methods. It introduces key formal languages i.e. Dafny, Ada/SPARK, Frama-C, and KeY alongwith the interactive theorem provers (Coq, Isabelle / HOL and Lean). The integration of Copilot and formal methods is proposed through development of IDE containing language servers. The examples of such existing IDEs are VSCode and Eclipse where multiple programming languages support is available in one IDE through Language Server Protocol (LSP).

A very comprehensive study on the state of the art of formal specification and verification on autonomous robotic systems is [56]. The authors did literature review on cyber-physical systems, omitting pure software-based systems. The survey covered human-controlled and autonomous systems. These were either remotely operated or self-governing. Formal properties like safety, security and reliability were considered. Mechanical and physical considerations are excluded from the scope of the survey [56].

[14] is a comprehensive work on NLP verification, where existing verification approaches are exhaustively analysed in the paper. A structured **NLP Verification Pipeline** has been established compromising six critical components: data selection, generation of perturbations, choice of embedding functions, definition of subspaces, robust training, and verification through existing algorithms. To validate this pipeline, ANTONIO tool has been implemented. It enabled modular experimentation with various pipeline components. The key contribution of [14] is the identification of gaps in existing approaches and the proposal of novel solutions to improve the robustness and reliability of NLP verification pipelines. NLP verification results have been reported through additional criteria. Standard verifiability metrics has been extended, comparing geometric with sematic subspaces. Semantic perturbations are employed while conducting experiments. In [14], the importance of reporting volumes, generalisability, and embedding error of verified subspaces is emphasised. The reason behind is the great impact of these factors on reliability and interpretability of verification results. The paper [14] indicates the future expansion by evaluating model robustness against adversarial perturbations and dataset variations.

Granberry et al. [34] explored how combining large language models (LLMs) with symbolic analysis can help generate specifications for C programs. They enhanced LLM prompts using outputs from PathCrawler and EVA to produce ACSL annotations. Their findings showed that PathCrawler generated context-aware annotations, while EVA contributed to reducing runtime errors.

The purpose of Dafny is to automate proofs by outsourcing them to an SMT solver. The SMT solver needs assertions while automating the process. [61] presented a framework named Laurel to generate Dafny assertions using LLMs. Mugnier et al. [61] designed two domain-specific prompting techniques. First one locates the position in code where assertion is missing. This is done through analysis of the verifier's error message. At the particular location with missing assertion, a placeholder is inserted. Second technique involves provision of example assertions from codebase. Laurel was able to generate over 50% of the required helper assertions, making it a viable approach to deploy, while automating program verification process.

The work [57] represents a novel framework named SpecGen to generate specifications through LLMs. Two phases are applied. First phase is about having prompts in conversational style. Second phase is deployed where correct specifications are not generated. Here, four mutation operators are applied to ensure the correctness of the generated specifications. Two benchmarks i.e. SV-COMP and SpecGen are used. Verifiable specifications are generated successfully for 279 out of 384 programs, making [57] a viable approach. The work [13] deals with the challenges involved in NL2SQL transformation, being widely deployed in Business Intelligence (BI) applications. Bora et al. [13] developed a

new benchmark focused on typical NL questions in industrial BI scenarios. Authors added question categories in the developed benchmark. Furthermore, two new semantic similarity evaluation metrics are represented in [13], increasing NL2SQL transformation capabilities.

4 TRACEABILITY OF SOFTWARE REQUIREMENTS

A dynamic requirements traceability model is proposed in [76], enabling improved software quality through verification and validation of functional requirements. The model ensured software scalability as well, dealing both small and large-sized projects. A novel model for traceability and verification in the early development phase is presented in [77]. Adaptability to requirement changes is improvised in the model and its impact is assessed. [29] provided a comprehensive review of software requirements traceability, covering key elements, challenges, and techniques. The study classified traceability approaches and highlighted prospects for future research.

The empirical analysis of requirements completeness is performed in [75]. By enforcing completeness in requirements ensured reduced defect rates. Here, traceability metrics is produced and regression analysis is performed to quantify the software quality. [40] introduced the RETRO tool for automating requirements traceability matrix (RTM) generation. The study showed that RETRO significantly improved accuracy and efficiency compared to manual tracing methods. [84] proposed a hybrid approach combining VSM and BTM-GA to enhance traceability link generation. The method outperformed traditional IR techniques, improving recall and precision, particularly in agile development contexts.

Trustrace [2] is a trust-based traceability recovery approach. It leverages mined data from software repositories. In comparison to standard IR techniques, Trustrace improved precision and recall in traceability link retrieval. [37] applied deep learning for traceability, incorporating semantic understanding and domain knowledge. This is done using BI-GRU. It outperformed traditional approaches of VSM and LSI in terms of accuracy and effectiveness. A topology-based model-driven approach for backward requirements traceability is presented in [6], formalising specifications and establishing trace links between real-world functional units and software artifacts.

In order to manage software evolution process, [16] proposed an event-based traceability mechanism. The work reported performance improvement in change management by linking artifacts through an event service. It maintained consistency being deployed in distributed development environment. Tracebok [19] is a body of knowledge on software requirements traceability. The framework categorised traceability approaches and provided guidance for implementing traceability in software projects. [58] reviewed visualisation tools and techniques for software requirements traceability. The challenges emphasised were scalability and visual cluttering, providing insights of improved traceability visualisation.

In [78], Z notation is used to represent the SRS and design artifacts in order to establish the traceability of functional requirements. The SRS document originally used UML diagrams for requirement analysis and software design. Z notation established trace paths based on defined rules. A prototype framework based on XML is developed to trace the requirements in software design. The designed framework is named as RVVF (Requirement Verification and Validation Framework). [15] established the concept that terminology extraction can improve traceability from formal models to textual requirements. Cerbah et al. [15] presented a fully implemented system that analysed text corpora to generate hierarchically organised terminological resources and formal class hierarchies. These resources and formal class hierarchies are then communicated to the Troeps knowledge server via an XML stream. The system also enabled user validation of terminology elements and supports bidirectional linkage between the model and source documents. Survey paper [82] examined requirements traceability, including definitions, challenges, and tools. [71] represented a tool

named TOOR - traceability of object-oriented requirements. The tool is based on the principles of hyper-programming and hyper-requirements.

Goknil et al. [33] addressed requirements and their relationships from a traceability perspective. It introduced a metamodel for requirements that included formally defined relation types. The relations are formalised using first-order logic to enable inference and consistency checking. A supporting tool demonstrated the approach on a real-world requirements document, helping to uncover hidden dependencies and detect contradictions.

5 FORMAL METHODS AND TESTING

A detailed examination of the interaction between formal specification techniques and software testing is presented in the 2009 ACM survey [41]. The paper argues that formal descriptions of systems can significantly support the testing process. It introduces a classification of formal specification languages into several distinct categories. These include model-based methods such as Z and VDM, languages grounded in finite-state representations like FSMs and Statecharts, algebraic approaches such as OBJ, process algebraic notations like CSP and CCS, and hybrid methods that integrate both continuous and discrete system behaviours—although the latter are treated as outside the scope of the survey.

In practical applications, formal methods can contribute to testing by either enabling the automated generation of test cases or offering mechanisms to define precise test oracles. Executable specifications, in particular, may be subjected to model-checking techniques to assess conformance with desired properties. The theoretical underpinnings that link formal specifications with testing processes are also explored, with [28] offering foundational contributions—especially in identifying test selection assumptions that underpin the effectiveness of test generation. Each category of formalism is associated with its own techniques for producing relevant test artefacts.

5.1 Model-Based

In this category, testing often involves partitioning the input domain using assumptions about uniform system behaviour within each partition. Logical expressions—frequently cast in disjunctive normal form—are used to define these partitions and guide automation. Further approaches include domain analysis techniques that focus on identifying critical boundaries for functions and operators. Additionally, refinement-based testing and mutation techniques are discussed as part of this landscape. Although highly suitable for defining test oracles, model-based methods are often limited in their ability to automatically generate tests without the assistance of theorem-proving tools.

5.2 Finite State-Based

Testing approaches that rely on finite-state representations frequently define correctness in terms of language-based conformance between the implementation and its specification. A common strategy involves using a fault model to constrain the number of system states considered. Test generation techniques initially focus on deterministic finite-state machines, before expanding to address partial and non-deterministic forms. While these methods offer structured ways to derive test suites, the primary challenge lies in managing the combinatorial growth in possible state sequences.

5.3 Process Algebras

Systems described using process algebra are typically interpreted through labelled transition systems (LTS), which can be infinite. To address this, state space reduction techniques are often employed. In many respects, the methods and challenges of testing in this domain resemble those found in finite-state approaches, particularly with regard to scalability and complexity management.

5.4 Algebraic

Algebraic specifications are especially well-suited to object-oriented software. Test cases in this setting may be derived either from the syntactic structure of operations or from the logical axioms that define their intended behaviours. This method provides a strong formal basis for validation, although transforming abstract axioms into executable test procedures requires further elaboration and interpretation.

The survey places particular emphasis on the value of automated reasoning tools in supporting test activities. Model checkers are highlighted as being capable of producing counterexamples when temporal properties are not satisfied, which can subsequently be re-purposed as test cases. Similarly, properties defined in temporal logic can guide the construction of structured test sequences.

5.5 Formal Tools

Isabelle/HOL is a powerful theorem prover based on higher-order logic. It features robust automation, a large repository of verified theorems, and tools for interactive proof construction. The system allows formal reasoning about complex mathematical properties and software systems, offering both depth and flexibility in its approach.

Frama-C is a modular platform designed for the formal analysis of C code. It supports ACSL (ANSI-C Specification Language) for specifying expected behaviour and includes plug-ins for static analysis, verification, and integration with theorem provers. This makes it highly applicable to domains that demand rigorous validation of software correctness.

There has been significant progress in the use of probabilistic techniques for formal verification [25, 45, 49]. Probabilistic model-checking focuses on evaluating how likely a system is to meet certain criteria, rather than delivering absolute verdicts. A comprehensive overview of developments in this area is provided in [1], particularly in the field of statistical model-checking, which offers scalable solutions for analysing stochastic systems in practical contexts.

Promela is a modelling language aimed at the representation of concurrent systems. It is paired with the SPIN model checker, which is widely used in both academic and industrial settings. A supporting tool, Modex, can automatically extract Promela models from C code. SPIN evaluates properties defined in linear-time temporal logic (LTL), and when verification fails, the counterexamples it provides can be used to create test cases. Its command-line interface makes it suitable for automation and integration into larger tool-chains.

TLA+ offers a mathematical framework for specifying systems, particularly those involving concurrency. It emphasises logical precision using simple mathematical constructs. PlusCal provides a more familiar, algorithm-like syntax that compiles directly into TLA+, enabling a smoother transition for developers accustomed to pseudocode-style representations.

6 FORMAL PROOFS IN UTP AND THEORY OF INSTITUTION

[88] provided a tutorial introduction to Hoare and He's Unifying Theories of Programming (UTP) and the concept of designs. It explained how alphabetised relational calculus could describe various programming constructs, illustrating their application to imperative programming theories like Hoare logic and the refinement calculus. [32] introduced the concept of institutions as a formal framework to model logical systems. It presented several foundational results, such as gluing signatures, preserving theory structuring, and extending institutions to include constraints for abstract data types, contributing significantly to the theory of specifications and programming languages.

[87] discussed Circus, a concurrent language that integrated imperative programming, CSP, and Z through the unifying theories of programming. It provided a formalisation of Circus in the UTP framework, highlighting its use for

refining concurrent systems. [39] explored integrating runtime verification into an automated UAS Traffic Management system. It demonstrated how runtime verification could ensure system safety by applying formal requirements to various subsystems, validated through real-world flight simulations.

[12] presented formal verification applied to the RTEMS real-time operating system, using Promela models and the SPIN model-checker to verify multi-core processor qualification for spaceflight. It discussed linking UTP semantics to enhance the test generation process, with a focus on future research directions.

[26] introduced Isabelle/UTP, an implementation of Hoare and He's UTP for unifying formal semantics. It enabled mechanising computational theories across paradigms and provided proof tools for Hoare logic, refinement calculus, and other computational paradigms, supporting the development of automated verification tools. [46] discussed the use of UML and the UML Testing Profile (UTP) in model-based testing for resource-constrained real-time embedded systems. It addressed the generation of test artefacts from UTP standards and presented a detailed algorithm for creating test cases for such systems.

[27] presented a Java model of the priority inheritance protocol in the RTEMS real-time operating system, verified using Java Pathfinder. It detected and fixed known bugs in the RTEMS implementation, ensuring the absence of issues like data races, deadlocks, and priority inversions. [83] proposed an iterative prompting framework for pre-trained language models to handle multi-step reasoning tasks. It introduced a context-aware prompter that dynamically synthesised prompts based on the current step's context, improving the model's reasoning capabilities in complex tasks.

7 FUTURE DIRECTIONS

In this section, we outline prospective directions informed by the literature review. Much of the literature in this review employed queries containing a problem description and some instructions to achieve a desired outcome. Such querying of LLMs without training or examples of the current task is typically referred as zero-shot prompting and shows excellent performance on many tasks [48]. Surprisingly, they also showed that the performance of LLM on some challenging problems can be improved by encouraging the LLM to reason using intermediate steps through a simple addition to problem prompts ("Lets think step by step").

7.1 Advanced Prompt Engineering

Beyond this approach is one-shot prompting that includes an example of a solved problem to guide the LLM into generating the desired output [55]. This can be extended to few-shot prompting where a number of differing examples guide the LLM. But improved results are not assured as some studies e.g. [95] show that zero-shot can outperform the few-shot case [96]. [42] reviewed the evolution of prompt engineering in LLMs, including discussions on self-consistency and multimodal prompt learning. It also reviewed the literature related to adversarial attacks and evaluation strategies for ensuring robust AI interactions.

Chain of Thought (CoT) [85] prompting involves a sequence of prompts producing intermediate results that are generated by the LLM and used to drive subsequent prompting interactions. These orchestrated interactions can improve LLM performance on tasks requiring logic, calculation and decision-making in areas like math, common sense reasoning, and symbolic manipulation. CoT requires the LLM to articulate the distinct steps of its reasoning, by subdividing larger tasks into multi-step reasoning stages, acting as a precursor for subsequent stages. But the CoT approach may require careful analysis when used with larger LLM offering long input contexts. This is because of the lost-in-the-middle problem where LLM show a U-shaped attention bias [42] and can fail to attend to information in the middle of the context window.

PromptCoT [94] enhanced the quality of solutions for diffusion-based generative models by employing the CoT approach. The computational cost is minimised through adapter-based fine-tuning. Prompt design is explored in detail in [3]. It discussed Chain-of-Thought and Reflection techniques, along with best practices for structuring prompts and building LLM-based agents.

Besta et al. [10] introduced the concept of reasoning topologies, examining how structures such as Chains, Trees, and Graphs of Thought improve LLM reasoning. They also proposed a taxonomy of structured reasoning techniques, highlighting their influence on both performance and efficiency. Structured Chain-of-Thought (SCoT) prompting was proposed by [53] to enhance code generation by incorporating structured programming principles. This approach significantly improved the accuracy and robustness of LLM-based code synthesis compared to standard CoT methods. Building on the theme of automation, [79] introduced Automate-CoT, a technique for automatically generating and selecting rational chains for CoT prompting. By minimising dependence on human annotations, it enabled more flexible adaptation of CoT strategies across diverse reasoning tasks. Complementing these efforts, [86] presented a prompt pattern catalog that offered reusable design patterns to optimise LLM interactions, thereby refining prompt engineering practices for a wide range of applications. Additionally, [90] proposed Reprompting (Gibbs sampling-based algorithm) for discovering optimal CoT prompts. The proposed prompting technique consistently outperformed human-crafted alternatives and demonstrated high adaptability across various reasoning benchmarks.

Retrieval Augmented Generation (RAG) [52] supplements problem information with specifically retrieved information and is often used in knowledge intensive tasks. This helps ensure the LLM attends specifically to the retrieved information when addressing the users prompt. LLM model selection (chat vs reasoning) and fine tuning such as with LoRA [43] remain among a growing number of possibilities for exploration.

Based on the literature survey conducted, we sketch one line research agenda: in VERIFAI, we aim to improve the techniques that bridge the gap between informal natural language description and rigorous formal specifications, through refinement of prompt engineering, the incorporation of chain-of-thought reasoning and the development of hybrid neuro-symbolic approaches.

8 CONCLUSIONS

The role of large language models in formalising software requirements is surveyed in this paper. The key contribution of the selected papers is on bridging the gap between informal natural language descriptions and rigorous formal specifications. We can enhance requirement formalisation through automating translations. The accuracy of translated ones is of key importance. We can deploy iterative refinements to improve the accuracy and correctness of the generated requirements. While LLMs is contributing significantly in improving development cycle, the challenges of ambiguity resolution, verification and domain adaptation shall be the focus areas of future research.

In future research, the refinement of prompt engineering techniques, the incorporation of chain-of-thought reasoning and the development of hybrid neuro-symbolic approaches are the key areas to look at. Our survey concludes with the remarks that the better collaboration with industry and academia will further enhance the domain.

9 ACKNOWLEDGEMENTS

This work is partly funded by the ADAPT Research Centre for AI-Driven Digital Content Technology, which is funded by Research Ireland through the Research Ireland Centres Programme and is co funded under the European Regional Development Fund (ERDF) through Grant 13/RC/2106 P2.