Translating Natural Language Requirements to Formal Specifications: A Study on GPT and Symbolic NLP

Iat Tou Leong*, Raul Barbosa†
University of Coimbra, CISUC, Department of Informatics Engineering
Email: *itleong@dei.uc.pt †rbarbosa@dei.uc.pt

***Abstract*—Software verification is essential to ensure dependability and that a system or component fulfils its specified requirements. Natural language is the most common way of specifying requirements, although many verification techniques such as theorem proving depend upon requirements being written in formal specification languages. Automatically translating requirements into a formal specification language is a relevant and challenging research question, because developers often lack the necessary expertise. In our work we consider the application of natural language processing (NLP) to address that research question. This paper considers two distinct approaches to formalise natural language requirements: a symbolic method and a GPT-based method. The two methods are evaluated with respect to their ability to generate accurate Java Modeling Language (JML) from textual requirements, and the results show good promise for automatic formalisation of requirements.**

***Index Terms*—Software engineering, Formal specification, Software verification, Java Modeling Language**

I. Introduction

The proliferation of software has raised concerns about dependability, particularly in critical fields like healthcare, aerospace, railway, automotive, among others. To ensure software dependability, verification and validation methods are employed once software is developed. However, these methods necessitate formal requirements to accurately assess software correctness. Static checking typically requires formal specifications and dynamic testing usually requires assertions written in rigorous languages.

Formalising requirements requires expertise in formal specification languages, which have rigorous syntax and semantics. However, not all software developers possess this expertise. Consequently, requirements are frequently composed in natural language. It is therefore a crucial aspect of V&V to translate those requirements into a specification language.

When verifying software correctness, natural language requirements must be translated into formal requirements. However, words can have different meanings depending on their usage. For example, the word "prime" can refer to "supremacy" in everyday conversation and to a number without factors other than one in mathematics. Errors in interpreting meanings can result in disastrous consequences. Furthermore, manually translating requirements requires effort and expertise.

This paper addresses the challenges of applying NLP to automatically translate natural language requirements into formal requirements in JML [1]. We consider two distinct NLP approaches: 1) obtaining JML translated using GPT [2] via ChatGPT; 2) a symbolic approach designed by the authors in which a compiler translates higher-order logic meaning representations, generated from natural language requirements using ccg2lambda [3], into formal requirements in JML. The two translations are evaluated by using OpenJML [4] with z3 [5] to check both the syntax and semantics correctness of the generated JML, as well as the correctness of programs with the generated JML. The overall goal of this paper is to study existing NLP techniques as a means to save high time and costs in software development, by translating natural language requirements into formal requirements that are needed in deductive verification.

The remainder of the paper is organised as follows: Section II briefly introduces the concepts related to this study. Section III provides explanations of the two approaches. Section IV provides a case study to the approaches. Section V surveys the related concepts and literatures. Section VI presents the conclusion.

II. Background

Automatically translating natural language requirements into formal requirements involves using NLP to translate textual requirements into a formal specification language.

A. Formal Specification Languages

A formal specification language is a language used to describe software in a precise, unambiguous and machine-readable way. Often, verifying the correctness of formally specified software can be achieved with deductive verification using theorem proving, among other techniques. JML specifications are written in code comments annotating Java programs. Method behaviour specifications start with `//@` followed by keywords like `requires` for preconditions and `ensures` for postconditions. Quantifiers (*i.e.*, `forall`, `exists`) and boolean expressions can be used to specify behaviour, such as checking for sorted values in an array. There is wide support for JML, including tools like OpenJML [4] and ESC/Java [6] that support deductive verification.

B. Natural Language Processing

Natural language processing is a research area aiming to develop computational models that can understand and interpret human language. NLP can be broadly categorized

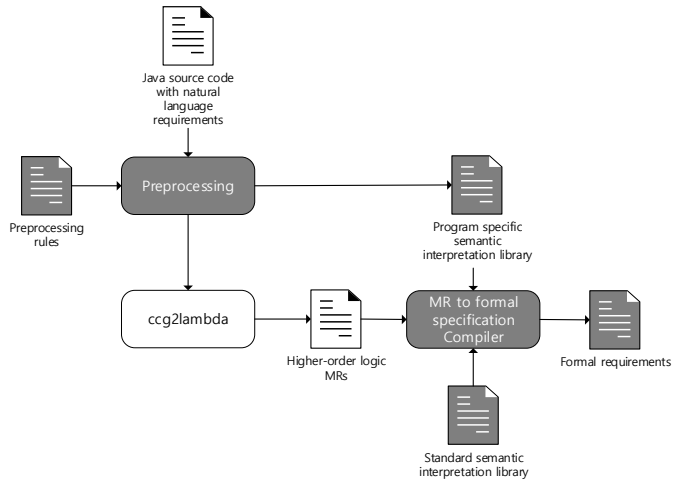

Fig. 1. Symbolic approach to automatic formalisation of textual requirements. Higher-order logic meaning representations are constructed through CCG derivation and our compiler translates those into JML.

into two approaches based on how it processes language: symbolic and deep learning. Symbolic NLP uses explicit representations, ranging from simple regular expressions to complex grammars, and algorithms to interpret the syntax and semantics of a language [3]. Deep learning NLP employs deep neural networks to learn the patterns and structures of language from large amounts of data [2].

III. Approaches

To produce formal requirements in JML, the symbolic approach is based upon a compiler designed by the authors and the deep learning approach is based upon GPT.

A. Symbolic approach

We have constructed a compiler that takes as meaning representation (MR) derived from input textutal requirements in code comments and follows the approach depicted in Figure 1. Text is preprocessed with extensible rules to replace words which might be incorrectly classified by the NLP tool. For instance, Java keywords representing literals (*e.g.*, true, false, null) are replaced by 'a [keyword]_value' (*e.g.*, a true_value). Such words are usually tagged as adjectives by the part-of-speech (POS) tagger, but they should be tagged as nouns because they are commonly used in boolean and object reference comparisons.

Each preprocessed sentence undergoes a translation process using ccg2lambda [3], which involves tokenization, combinatory categorical grammar (CCG) tree derivation, and semantic composition. A sentence is tokenized using white spaces as delimiters, and then tagged as well as parsed by the C&C parser to generate a CCG tree. Once the leaves and nodes are matched with the rules in the template written in NLTK [7] lambda calculus format, semantics are given to the words in the nodes, resulting in a higher-order logic MR. To enhance the MR, we annotate all MR predicates with their respective syntactic categories.

A compiler, which we named Meaning Representation Compiler (MEARC), has been designed and implemented for translating an MR to its corresponding JML (publicly available at https://github.com/itlchriss/MEARC). The core concept of the translation is to match predicates with semantic interpretations (SIs) using their syntactic categories and arities. Each SI is a JML construct that accepts arguments to form a JML expression. The translation starts with building an abstract syntax tree (AST) by parsing the MR with an LR(1) grammar. Internal nodes are either connectives (*i.e.*, and(&), or(|), *etc.*) or predicates. Leaf nodes are variables that are arguments of the predicates.

An analysis is applied to all predicate nodes to match their SI in an extensible SI library. If a predicate node has no SI matched, and it is being an argument to another predicate, an error is thrown to signal an SI must be provided. After mapping the SIs to predicates, JML constructs are synthesised from subtrees by substituting leaf nodes to the arguments of the SI. Synthesised JML constructs are stored in the internal nodes, and the leaf nodes are removed. Finally, target JML is produced by post-order traversing the tree.

B. GPT approach

Training the GPT model [2] can be divided into two stages. The first stage is an unsupervised pre-training process on a large corpus of text without any explicit labeling. The second stage is a supervised fine-tuning process, which involves training the model on a labeled dataset to adapt it to a specific NLP task. To utilize the GPT model for translation, ChatGPT can be employed by inputting queries into the model. For each query, translation of a requirement to JML is explicitly asked for, and the JML is returned with some additional explanations.

IV. Evaluation and Discussion

To assess the approaches, we have written ten different requirements as input. For GPT, we used the query "Please convert this sentence to JML: [requirement]", where "[requirement]" is the textual specification. For both the symbolic approach and GPT, the resulting JML is checked for syntactic and semantic correcness using OpenJML [4]. Correctness with respect to the corresponding Java program is statically checked by OpenJML with the z3 [5] theorem prover. In the results that follow, a JML specification is considered correct only if it satisfies the syntax and semantics checks, as well as the verification checks performed by the z3 theorem prover.

Table I shows the outcomes for both approaches on the natural language requirements. The symbolic approach successfully produced correct JMLs in 7 out of 10 translations. In the 3rd and 5th rows the MRs were incorrectly generated, resulting in the absence of generated JML from the compiler. Further development of language interpretation is necessary to improve the MR derivation for such cases. Regarding row number 2, the generated JML compares the contents of two arrays, but the deep equals method in Java implements nested array comparison, leading to rejection by the theorem prover.

TABLE I
RESULTS OF TRANSLATING NATURAL LANGUAGE REQUIREMENTS TO JML USING SYMBOLIC NLP AND GPT.

Natural language requirement	JML generated by the symbolic approach	Correct	JML generated by GPT	Correct
1. The array should not be null.	requires (array != null);	Y	requires array != null;	Y
2. If parameter x deeply equals parameter y, then result is true.	ensures(((((x == null && y == null) && ((x == y) || (x.equals(y)) || Arrays.equals(x, y))) || ((x != null && y != null && x.length == y.length) && (\forall int i; 0 <= i < x.length; ((x[i] == null && y[i] == null) || (x[i] != null && y[i] != null && (!x[i].getClass().isArray() && !y[i].getClass().isArray() && x[i].equals(y[i])))))))) ==> (\result==true));	N	requires x.deepEquals(y); ensures \result == true;	N
3. If and only if parameter x deeply equals parameter y, then result is true.	–	N	requires x.deepEquals(y) <==> \result == true;	N
4. If the input n is a prime number, the result is true.	ensures((n == 2 || (n > 2 && (\forall int k; n > 2 && 2 <= k && k <= n/2; n % k != 0))) ==> (\result==true));	Y	requires isPrime(n); ensures \result == true;	N
5. If and only if the input n is prime, the result is true	–	N	requires isPrime(n) <==> \result == true;	N
6. If the result is true, the input number is even	ensures((\result==true) ==> (number % 2 == 0));	Y	requires isEven(n) ==> \result == true;	N
7. If the result is false, the input number is not even	ensures((\result==false) ==> (!(number % 2 == 0)));	Y	requires !isEven(n) ==> \result == false;	N
8. The input arr should be sorted in descending order	ensures (\forall int i; 0 <= i && i < arr.length - 1; arr[i] >= arr[i + 1]);	Y	requires (\forall int i, j; 0 <= i && i < j && j < arr.length; arr[i] >= arr[j]);	Y (?)
9. The result is less than or equal to 2000 and the result is greater than or equal to 0.	ensures((\result >= 0) && (\result <= 2000));	Y	ensures \result >= 0 && \result <= 2000;	Y
10. The input num1 is less than or equal to 1000 and the input num2 is less than or equal to 1000.	requires((num2 <= 1000) && (num1 <= 1000));	Y	requires num1 <= 1000 && num2 <= 1000;	Y

Regarding the GPT results, 4 out of 10 translations are fully correct, although quite a few others come close. In rows 2, 4, 5, 6 and 7, the translations are incorrect due to a method used in the specification not existing or its call being disallowed. This is likely due to GPT missing contextual information and causes a semantic error in the JML specification. Furthermore, the deepEquals method usage in row number 2 is syntactically incorrect (the method takes two arguments instead). At rows number 3, 5, 6, and 7 the translations have syntax errors due to referring to results in preconditions ('\result' is disallowed in 'requires' clauses). Overall, these findings indicate that GPT requires a specific training corpus to acquire the necessary knowledge, such as Java API specifications.

GPT's result at row number 8 is arguably correct. The condition body is correctly translated (*i.e.*, the \forall clause for 'sorted in descending order'). However, it is considered to be a precondition while it arguably should be a postcondition (the method does not refer to a'result', which hints to a postcondition). Without context, the requirement could be a postcondition of a sorting method or a precondition of a binary search implementation. This suggests a research opportunity to improve the deep learning approach by providing contextual information from programs. (*e.g.*, void return type of a pure method hints that it modifies its arguments).

To further investigate the issue of incorrect method calls in specifications, we modified the ChatGPT queries for requirements number 2 and 4 by adding the phrase "without calling a separate function". The results, shown in Table I, show that in both cases the generated JML correctly replaces 'requires' with 'ensures'. However, that was not our aim when modifying the queries. This exposes some non-determinism in the results provided by GPT (a known characteristic). Furthermore, GPT removes the method call inconsistently among the two queries. Method behaviour is correctly specified for 'prime', while the usage of 'deepEquals' remains in the first row of Table I.

Finally, comparing the two approaches, GPT is more accurate in interpreting the meaning of natural language sentences. However, it does not handle perfectly the characteristics of the Java API and language. Further training specifically using the corpus of the Java APIs would likely improve GPT with this respect. The results also indicate that contextual information from programs and idioms from software engineering play a crucial role in the translation process. Hence, although the symbolic approach is more limited in language interpretation, it provides more accurate JML whenever the language is correctly interpreted.

V. RELATED WORK

This paper focuses on automatically formalising natural language requirements. Several existing methods achieve this using restricted sets of natural language using structured models [8], [9] and internal predefined rules [10], heuristics with models [11], [12], based on analysis and heuristics [13], [14], redefining logic and its parser [15]. In contrast, the two approaches employed in this paper do not impose any intentional restrictions on the natural language input.

The symbolic approach presented in this paper is related to ARSENAL [16]. While ARSENAL is designed to generate linear temporal logic, the symbolic approach presented here

TABLE II
RESULTS OF MODIFYING THE QUERY TO CHATGPT. THE ORIGINAL QUERY IS "PLEASE CONVERT THIS SENTENCE TO JML: [REQUIREMENT]". THE MODIFIED QUERY APPENDS "WITHOUT CALLING A SEPARATE FUNCTION".

Natural language requirement	JML produced by original query	JML produced by modified query	Correct
If parameter x deeply equals parameter y, then result is true.	requires x.deepEquals(y); ensures \result == true;	ensures (\result == true && x.deepEquals(y)) || (\result == false && !x.deepEquals(y));	N
If the input n is a prime number, the result is true.	requires isPrime(n); ensures \result == true;	ensures (\result == true && (\forall int i; 2 <= i && i < n; n % i != 0)) —— (\result == false && (\exists int i; 2 <= i && i < n; n % i == 0));	Y

can handle higher-order logic. As a result, the input complexity of the two approaches differs significantly, as linear temporal logic does not support quantifications. Another key difference is that the symbolic approach generates JML specifications which may be verified using theorem proving.

An approach named FRET [17] formalises structured natural language requirements, with an industrial application demonstrating promising results. However, the requirements accepted by FRET does not support quantifications. It would be relevant to explore the use of GPT to convert unrestricted natural language requirements to FRETish requirements.

VI. CONCLUSION

Analysing software using natural language remains a challenging problem with relevant time and cost implications for any organization applying V&V techniques. To tackle this issue, a possible solution is to translate natural language requirements into formal requirements and then utilize deductive verification to assess the translated requirements.

This paper assesses the feasibility of two distinct methods for automatically formalising natural language requirements. The first method employs a symbolic approach that utilizes ccg2lambda [3] to translate meaning representations derived from natural language into formal requirements. The second method involves submitting queries to GPT [2]. Both methods are able generate JML specifications as the target language.

The paper evaluates both approaches on a set of requirements and the main conclusion is that both the symbolic approach and GPT provide meaningful and promising results. In terms of language comprehension, GPT outperforms the symbolic approach as it always interprets the meaning of the textual requirements. However, the symbolic approach is more accurate in generating JML when it is able to interpret the textual input and construct a meaning representation. Therefore, both approaches can make a significant contribution to automate the formalisation of natural language requirements, and thus reduce the effort of applying formal techniques.

REFERENCES

[1] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: A behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, p. 138, May 2006.

[2] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.

[3] P. Martínez-Gómez, K. Mineshima, Y. Miyao, and D. Bekki, "ccg2lambda: A compositional semantics system," in *Proceedings of ACL 2016 System Demonstrations*. Berlin, Germany: Association for Computational Linguistics, August 2016, pp. 85–90.

[4] D. R. Cok, "Openjml: Jml for java 7 by extending openjdk," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 472–479.

[5] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[6] D. R. Cok and J. R. Kiniry, "Esc/java2: Uniting esc/java and jml," in *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer, 2004, pp. 108–128.

[7] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. O'Reilly Media, Inc., 2009.

[8] I. S. Bajwa, B. Bordbar, and M. G. Lee, "Ocl constraints generation from natural language specification," in *2010 14th IEEE International Enterprise Distributed Object Computing Conf.*, 2010, pp. 204–213.

[9] C. Wang, F. Pastore, and L. Briand, "Automated generation of constraints from use case specifications to support system testing," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 23–33.

[10] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 242253.

[11] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 27–37.

[12] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, *C2S: Translating Natural Language Comments to Formal Program Specifications*. New York, NY, USA: Association for Computing Machinery, 2020, p. 2537.

[13] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: Bugs or bad comments?*/," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 145158.

[14] L. Tan, Y. Zhou, and Y. Padioleau, "Acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1120.

[15] C. Menghi, S. Nejati, K. Gaaloul, and L. C. Briand, "Generating automated and online test oracles for simulink models with continuous and uncertain behaviors," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 2738.

[16] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, "Automatically extracting requirements specifications from natural language," *CoRR*, vol. abs/1403.3142, 2014.

[17] M. Farrell, M. Luckcuck, O. Sheridan, and R. Monahan, "Fretting about requirements: Formalised requirements foranaircraft engine controller," in *Requirements Engineering: Foundation for Software Quality*, V. Gervasi and A. Vogelsang, Eds. Cham: Springer International Publishing, 2022, pp. 96–111.