The Journal of Systems and Software 211 (2024) 112009

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Translating meaning representations to behavioural interface specifications☆

Iat Tou Leong *, Raul Barbosa

University of Coimbra, CISUC, Department of Informatics Engineering, Portugal

ARTICLE INFO

Dataset link: MEARC

Keywords:
Software engineering
Formal specification
Software verification
Java modelling language
Compilers

ABSTRACT

Higher-order logic can be used for meaning representation in natural language processing to encode the semantic relationships in text. Alternatively, using a formal specification language for meaning representation is more precise for specifying programs and widely supported by automatic theorem provers, while deductive verification based on higher order logic is less common for mainstream programming languages. This paper addresses the research question of translating higher-order logic meaning representations generated from method-level code comments into a formal specification language that extends first-order logic. Doing so requires resolving possible ambiguities in determining the appropriate semantics for predicates. This is an open challenge in the path toward using natural language processing with formal methods. To address this, the paper proposes an approach and constructs a compiler for translating meaning representations, generated from Java programs with method-level comments, into Java Modelling Language. We evaluate the compiler on a set of representative benchmarks, including programs and specifications from the Java API, by generating Java Modelling Language specifications and statically checking them with a theorem prover. Results show that in 94% of the cases Java Modelling Language is accurately generated and in 97% of those cases it can be automatically checked with a state-of-the-art theorem prover.

1. Introduction

Formal specification languages allow one to unambiguously describe program behaviour. They provide predicates such as Boolean side-effect free expressions in the programming languages. A significant advantage of specification languages is the possibility of using existing verification tools, such as theorem provers, to automatically check the correctness of programs. The languages accepted by most of these tools build upon formal foundations such as first-order logic, *e.g.*, JML (Leavens et al., 2006), Object Constraint Language (Warmer and Kleppe, 2003), Dafny (Leino, 2010) and ACSL (Delahaye et al., 2013). Formal verification using these languages is supported by many existing tools, for instance, OpenJML (Cok, 2011), KeY (Ahrendt et al., 2005), Why3 (Filliâtre and Paskevich, 2013), Dafny, ESC/Java2 (Chalin et al., 2005) and frama-c (Kirchner et al., 2015). Nevertheless, there exists few works for HOL for formal verification (Slind and Norrish, 2008; Barbosa et al., 2019; Owre et al., 1992).

Advances in natural language processing allow one to automatically construct MRs, which are formal representations of the semantic relationships in textual input. In the context of this paper, we consider MRs of textual specifications of programs. A relevant research question consists in using such MRs to formally verify the correctness of programs. However, the formalisms adopted for MRs include higher-order logic (Martínez-Gómez et al., 2016) and abstract meaning representation (Banarescu et al., 2013). In addition, the semantics of these MRs provide insufficient support to the predicates that formal specification languages can express, for instance, nested quantification (Bos, 2016) and Boolean side-effect free expressions. Therefore, one cannot use MRs with existing automatic theorem provers because most existing implementations accept specification languages related to first-order logic.

This paper addresses the research challenge of automatically translating MRs, derived from natural language specifications, into formal specifications. Such a challenge is fundamental for applying NLP in conjunction with formal methods, for instance, to formally verify that programs fulfil their textual requirements specified in code comments. Despite recent advances in NLP, this remains a challenge and an open research question, due to ambiguities resulting from unrepresented syntactic categories, multiple dictionary meanings of words, domain-specific meanings and predicates used as arguments to other predicates.

We design and implement a compiler that takes MRs generated from method-level comments, using established NLP techniques, and translates them into JML specifications. JML is a specification language for Java (Chalin et al., 2005) which combines design by contract (Meyer,

☆ Editor: Laurence Duchien.
* Corresponding author.
E-mail addresses: itleong@dei.uc.pt (I.T. Leong), rbarbosa@dei.uc.pt (R. Barbosa).

https://doi.org/10.1016/j.jss.2024.112009
Received 10 March 2023; Received in revised form 12 October 2023; Accepted 18 February 2024
Available online 22 February 2024

1997) with model-based specification (Jones, 1986). The resulting JML specifications can be used to verify the correctness of Java programs by means of techniques such as static checking and dynamic analysis (*e.g.,* theorem proving and software testing).

The overall goal is to bridge the gap between NLP and formal specification languages. This is divided into two research questions: (RQ1) we hypothesise that it is feasible to enhance NLP-based construction of MRs using an extensible knowledge base that contains programming language-specific idioms and semantic interpretations (SIs) of predicates; and (RQ2) we hypothesise that it is possible to design a compiler to synthesise JML specifications from MRs, by resolving ambiguities through a combination of symbolic information from the program and the NLP process. In more detail, this work makes the following contributions:

- *Extension of an existing NLP approach with a rigorous knowledge base (Section 4.1).* We apply an NLP method that generates MRs from Combinatory Categorical Grammar (CCG) derivation trees (Martínez-Gómez et al., 2016). We observe that this method lacks the ability to identify software engineering idioms and knowledge of programming languages. Because the NLP parser cannot recognise such idioms and knowledge, MRs cannot be reduced correctly for the purpose of formal specification. To address this, we design an extensible knowledge base consisting of *idioms* and single-word terms relating to software engineering and programming languages, and *SIs* for the predicates that appear in the MRs.
- *Design and implementation of a compiler that takes MRs, along with contextually relevant information from the NLP, and converts them into JML contracts (Section 4.2).* The compiler is restricted to the Java language, method-level comments, for non-concurrent programs. We annotate predicates in the MRs, generated from CCG derivation trees, with the syntactic categories of the predicates. The compiler takes an annotated MR, constructs an AST and applies rule-based transformations and SIs of predicates (tree nodes) to emit a JML specification.
- *Experimental evaluation of the compiler is performed by checking the correctness of the JML generated by it (Section 5 & (Section 6)).* Representative methods from the standard Java API, along with their official documentation, were given as input to the compiler, and the resulting JML specifications were deductively verified with OpenJML. Static checking substantiates the accuracy and usefulness of the compiler.

Overall, we believe that the proposed approach, along with the practical implementation, may be useful to facilitate formal reasoning with natural language, to improve the conversion of MRs to useful JML specifications, and to improve program analysis by means of rigorous specifications, both for static checking (which is the focus of this paper) and other techniques such as software testing.

The remainder of this paper is organised as follows. Section 2 describes related concepts and background. Section 3 identifies the main sources of ambiguity affecting MRs. Section 4 presents the proposed approach for translating MRs to JML specifications. Section 6 discusses the empirical findings and the main observations. Section 7 discusses the threats to validity of the proposed approach. Section 8 provides a survey to the related works in the field. Section 9 presents the conclusion.

2. Background

The aim of our approach is to bridge the gap between meaning representations (MRs) and formal specification languages. To this end, we apply NLP to extract MRs and translate those representations into a formal specification language. Therefore, background knowledge of specification languages and NLP is necessary for the remainder of the paper.

2.1. Formal specification languages

Formal specification languages serve the purpose of describing requirements with formal syntax and semantics. Checkable specifications written in formal specification languages allow developers to have better insights and understanding of the software requirements and design. Moreover, debugging can also be achieved with tools built to support such languages. By using formal specification languages, one may also use deductive verification to prove that the software conforms to its specification, as is the case of theorem proving. The JML (Leavens et al., 2006), Object Constraint Language (Warmer and Kleppe, 2003), Dafny (Leino, 2010) and ACSL (Delahaye et al., 2013) are languages for describing the behaviour of software, where ACSL is inspired by JML.

The JML specifications are written in the *comments* in the source code files starting with /*@ or //@. Java developers can adopt JML easily because most of the Java syntax can be applied in JML. The basic features include using explicit keywords of *requires*, *ensures*, *invariants* to represent the construct of preconditions, postconditions and invariants. Boolean expressions always follow these clauses. In addition, *result* keyword specifies the reference of the returning value of a function. Such a feature can increase the level of separation of concern as developers do not have to specify the return variable in the contracts. Burdy et al. suggest many advantages of using JML (Burdy et al., 2005). First, JML helps to increase the readability of predicates as the syntax is closer to Java. Besides, JML can be written in the comments as part of the source code as well as the formal model. This feature helps to fuse the gap between the formal model and the source code. Finally, JML has wide support of tools such as ESC/Java2 (Cok and Kiniry, 2004), and OpenJML (Cok, 2011), which allows JML to be convenient in software testing.

Using JML, one can specify method preconditions and postconditions, class invariants, model-based behaviour, and exceptional behaviour. Preconditions and postconditions are specified using quantifiers over Java boolean expressions, and JML includes the necessary keywords (`requires`, `ensures` and quite a few others) for describing the behaviour of Java programs expressively. There also exists informal expressions (Leavens et al., 2006) in JML specifications (see Fig. 1 at line 1) which accepts comments in natural language, where such specifications are always assumed to be *true* in verification. For simplicity, we refer these specifications as *informal specifications* throughout the paper.

Our approach translates such informal specifications into JML, thereby supporting the possibility of using an SMT solver such as z3 (de Moura and Bjørner, 2008a), to enable static checking of properties. Dynamic testing, based upon runtime assertion checking, is also a possibility to consider in future work.

2.2. Natural language processing

NLP is a subfield of artificial intelligence and linguistics that studies the use of computers for analysing human language. Various kinds of meaning representations are derived to represent natural language sentences through lexical, syntactic and semantic analysis. Existing tools such as NLTK (Bird et al., 2009), CoreNLP (Manning et al., 2014), OpenNLP (Schmitt et al., 2019) and ccg2lambda (Martínez-Gómez et al., 2016) generate different types of MRs by syntactic parsing. MRs are the formal structures representing the meaning of linguistic expressions. The main principle is that two sentences with different meanings should have different MRs. Examples of MRs are lambda expressions (Martínez-Gómez et al., 2016) and abstract meaning representations (Banarescu et al., 2013). These MRs use higher-order logic (HOL) encoding. CCG (Steedman, 1996) is a type-driven grammar formalism that is suitable for characterising natural language syntax. Each CCG entry consists of a syntactic category which defines valency (the number of grammatical elements can be combined) and directionality (backward or forward). Some example basic categories are S(Sentences), N(Nouns) and NP(Noun Phrases).

```
1  //@ requires (*The input arr should not be null*);
2  //@ ensures (*The input arr should be sorted in ascending order*);
3  public static void sort(int[] arr){
4      ...
5  }
```

Fig. 1. Sort method documented with JML specifications using informal expressions at line 1 and 2.

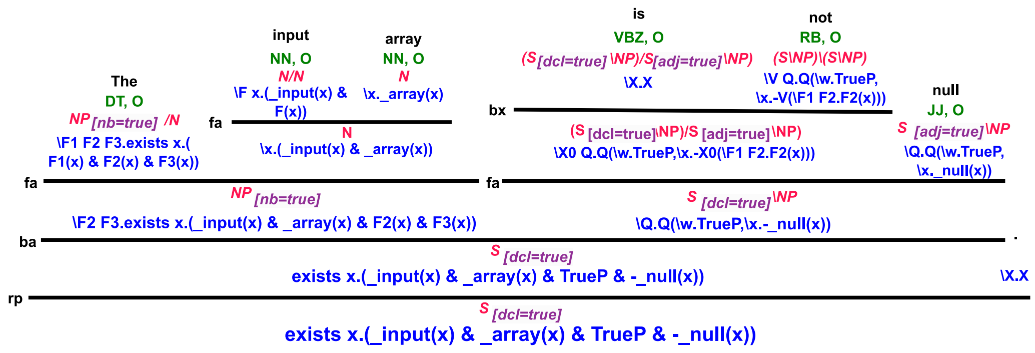
The
DT, O
NP[nb=true] /N
\F1 F2 F3.exists x.(F1(x) & F2(x) & F3(x))
input
NN, O
N/N
\F x.(_input(x) & F(x))
array
NN, O
N
\x._array(x)
fa
N
\x.(_input(x) & _array(x))
is
VBZ, O
(S[dcl=true]\NP)/S[adj=true]\NP)
\X.X
not
RB, O
(S\NP)\(S\NP)
\V Q.Q(\w.TrueP, \x.-V(\F1 F2.F2(x)))
bx
(S[dcl=true]\NP)/S[adj=true]\NP)
\X0 Q.Q(\w.TrueP,\x.-X0(\F1 F2.F2(x)))
null
JJ, O
S [adj=true]\NP
\Q.Q(\w.TrueP, \x._null(x))
fa
NP[nb=true]
\F2 F3.exists x.(_input(x) & _array(x) & F2(x) & F3(x))
fa
S[dcl=true]\NP
\Q.Q(\w.TrueP,\x.-_null(x))
ba
S[dcl=true]
exists x.(_input(x) & _array(x) & TrueP & -_null(x))
.
\X.X
rp
S[dcl=true]
exists x.(_input(x) & _array(x) & TrueP & -_null(x))

Fig. 2. CCG derivation for the example sentence *"The input array is not null"*. Each word in the sentence has a pre-defined lambda expression representing its semantics. The black lines indicate reductions are applied on the lambda expressions above it. The results of these reductions are produced under these black lines. MR is generated after all the reductions are applied.

Categories are combined by using combinatory rules to form derivations. There are two basic rules: forward and backward compositions. Forward composition is also known as type raising. For instance, in Fig. 2, the entry *input* with rule N/N indicates that it accepts a noun on its right to form a composite noun, such that it combines with *array*. The determiner *the* is combined with the composite noun to form a noun phrase. These two operations are forward compositions. Similarly, *is* looks for an entry with S\NP to form a verb phrase. The entry *not* fits the argument type and turns it into a verb with negative, then to forward look for *null* to form the verb phrase. Notice that the verb phrase with the type $S_{[dcl]}\backslash NP$, which indicates that it looks for a noun phrase at its left to form a sentence, such an operation is an example of backward composition. Readers may have already noticed that rules may be combined in both directions. For instance, the term *input array* can be combined with the verb to form S\(S\NP) with the term *null* is derived as S\NP. This is so-called 'spurious' derivation, that informally states an unambiguous sentence can have multiple parses. The C&C parser (Curran et al., 2007) resolves this issue by defining a model with normal-form using only type-raising and composition, and a model with predicate-argument dependency probabilities. Therefore, the parser always returns only one derivation with the greatest probability score, and ignore the rest of the derivations.

Ccg2lambda (Martínez-Gómez et al., 2016) builds upon the C&C parser (Curran et al., 2007) that parses sentences according to the CCG rules to form the parsing trees, such that each CCG leaf (*i.e.,* a word) can be applied by a template written in NLTK semantics format. The lambda term in the template is applied to the based form of the word. HOL encodings can be found in generalised quantifiers (*e.g.,* most, half of), modals (*e.g.,* should, must), veridical and anti-veridical predicates (*e.g.,* manage, fail) and attitude verbs (*e.g.,* know, remember). Nouns (single and plural), adjectives and comparatives are constructed using first-order logic (FOL).

3. Ambiguity of meaning representations

We have analysed the MRs generated from the documentation of Java classes, to categorise the main sources of ambiguity that prevent direct translation into a formal specification language. To illustrate with one example, the specification "The array should not be null" is translated to MR using ccg2lambda in Fig. 2. Such specification is translated to MR "exists x.(_array(x) & TrueP & -_should(_null(x)))", which includes the four types of ambiguities found in our analysis:

Ambiguity 1 Many predicates can assume multiple syntactic categories depending on their usage, and, although part-of-speech (POS) is used, such information is not included in MRs. In the example, the word *array* can be a noun or a verb, so there can be two possible meanings based on the decision of the POS component. However, without this information, an MR may have several possible interpretations.

Ambiguity 2 Every word may have multiple dictionary meanings; thus, every predicate may have ambiguous interpretation. Multiple understandings might be constructed from a single MR. There are multiple meanings for the words *array* (5 meanings) and *should* (14 meanings), according to the Oxford dictionary. One may interpret the example MR into $(5 \times 14) = 70$ different understandings. Therefore, contextual and symbolic information from programs is needed for choosing the correct meanings.

Ambiguity 3 In higher-order logic MRs, predicates might be arguments of other predicates. The predicate *null*, which accepts a variable *x*, is an argument to the predicate "should". However, FOL does not allow predicates to be arguments of other predicates. There may be multiple orderings to resolve the semantic relationships, forming many different possibilities.

Ambiguity 4 Predicates might have domain-specific meanings, but POS contradicts such meanings. Words such as *null*, *true* and *false* are literals in the Java programming language. They are commonly compared with other references and expression results in the source code. Such words should be categorised as nouns, intuitively, but are tagged as adjectives. Therefore, programming language information is required for semantic interpretation of these words.

4. Approach

We aim to translate HOL MRs to a formal specification language, namely JML, which is closely related to FOL. Fig. 3 illustrates our approach for constructing JML method-level specifications. This paper addresses the steps in shaded colour. For clarity of presentation, in the remainder of this section, we use as a running example an informal JML specification for a sorting program, shown in Fig. 1. Our approach consists of two pipelines:

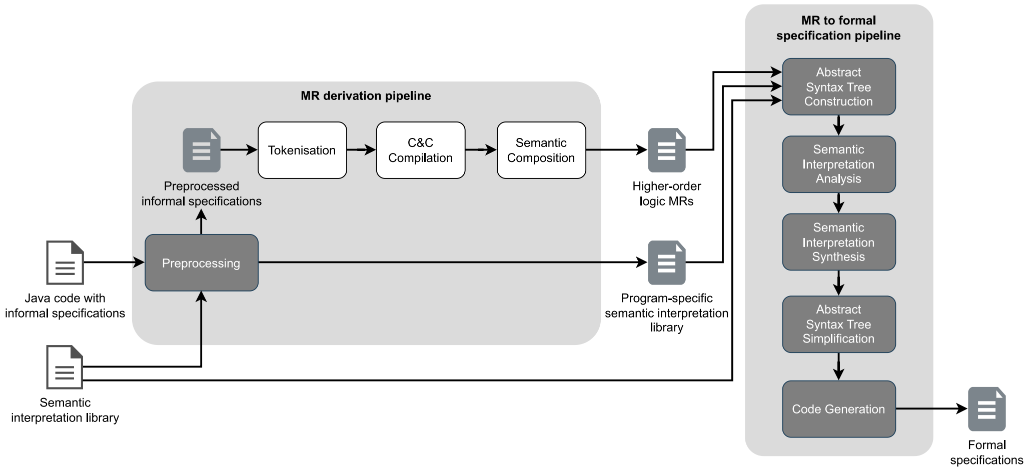
MR derivation pipeline
Preprocessed informal specifications
Tokenisation
C&C Compilation
Semantic Composition
Higher-order logic MRs
Java code with informal specifications
Preprocessing
Program-specific semantic interpretation library
Semantic interpretation library
MR to formal specification pipeline
Abstract Syntax Tree Construction
Semantic Interpretation Analysis
Semantic Interpretation Synthesis
Abstract Syntax Tree Simplification
Code Generation
Formal specifications

Fig. 3. An MR to formal specification translation diagram. The ccg2lambda pipeline generates MRs from informal specifications written in the comments in the Java source code. An MR to formal specification compiler translates the generated MRs to formal specifications. This translation requires the information from a standard semantic library and a program specific semantic interpretation library. This paper addresses the steps in shaded colour.

1. MR derivation (Section 4.1): The derivation of MRs from natural language specifications is organised into two steps:
   1.1. Preprocessing (Section 4.1.1): Natural language specifications in the code comments using requires and ensures clauses with informal expressions are extracted from the code. To resolve the ambiguities listed in Section 3, these sentences are preprocessed with the notion of idioms (sequences of words with specific meanings).
   1.2. Deriving MR from preprocessed sentences (Section 4.1.2): ccg2lambda (Martínez-Gómez et al., 2016) is modified to generate MR with all predicates annotated with their syntactic categories from natural language sentences. Derivation trees are obtained from parsing the preprocessed sentences with CCG, building upon ccg2lambda, and converted to MRs.
2. MR to JML translation (Section 4.2): A compiler translates MRs to formal JMLs. The translation process is organised into four steps:
   2.1. AST construction (Section 4.2.1): A flex and bison based parser constructs ASTs from MRs. A compile-time symbol table (CST) is constructed.
   2.2. SI analysis and synthesis (Section 4.2.2 & 4.2.3): Predicate SIs are identified with a CST from the library of SIs, the contextual information and annotations in the source code. Some predicates may have no SIs in the library and annotations. Analysis on these predicates is performed to decide if they can be removed (cf. 4.2.2). Whenever a predicate without SI cannot be removed, the translation is stopped and a symbol error is shown. Synthesis is applied to subtrees rooted at SI-identified predicates and operators.
   2.3. AST simplification (Section 4.2.4): The AST is simplified using rules to remove edges and nodes.
   2.4. Code generation (Section 4.2.5): Formal JML is generated through a recursive traversal of the AST.

The steps listed above show each respective sub-section in parenthesis. For clarity of presentation, in the remainder of this section, we use as a running example an informal JML specification for a sorting program, illustrated in Fig. 1. The source code is publicly available.

4.1. Meaning representation derivation

We begin with Java code with informal JML specifications, consisting of text delimited by (* *), which contains preconditions and postconditions specified by JML requires and ensures clauses, respectively. To translate such informal specifications into JML, the MRs of these informal specifications are required to provide syntactic meanings. To derive these MRs, informal specifications are preprocessed, and processed with ccg2lambda.

4.1.1. Preprocessing

Idioms and terms with specific meanings in software engineering are not recognised correctly in the MRs. These idioms and terms can be reused in several programs or used in a particular method specification. The preprocessing process assists the NLP in recognising these idioms as predicates by concatenating words with underscores (_). To specify these idioms, the compiler accepts two alternatives:

- The extensible standard SI library of SI contains commonly used idioms. The file is in the well known yaml format. Each entry of an idiom consists of the following:
  - term: a scalar value of idiom or single-word term, which is ready to be recognised as a predicate;
  - syntax: a sequence of syntactic categories;
  - interpretation: a scalar value of JML template/reserved keyword, which consists of variables that are arguments of this predicate;
  - arity: a scalar value specifying the number of arguments present in the template;
  - arguments: a sequence of argument variables required in the template. Asterisk (*) is used when the arguments do not affect the resulting SI.
  - argument types: optional. A sequence of argument variable types specifies that the SI can only be matched if these argument types are matched.

  In the current version, we provide a keyword '_sub'. This keyword provides a synthesis operation that can be applied to a node with two children, where one must be 'Synthesised' node and another must be 'Template' node. Synthesis operation (cf. 4.2.3) then substitutes the 'Synthesised' node SI to the 'Template' node SI arguments. For instance, a phrase *length of arr* should have a JML template '(x).length' such that a symbol from source code 'arr' is substituted to form the SI as 'arr.length'.
- SI annotations support idiom recognition for a single method. For instance, we have an SI annotation in the following format:
  ```
  //@ semantics "input arr", [NN], 1, [*], arr
  ```
  The above annotation denotes that when the idiom *input arr* is recognised as a noun, the arity must be one, and the SI is the symbol arr (see Fig. 1 at line 3). The annotation method complements if there is an absence of SI in the SI library. One can test an SI using the annotation method and extend the library for long-term usage.

This format allows users to add new elements to the SI library if needed. For instance, adding a predicate using a JML template with multiple syntactic categories and entries with different JML templates related to the same predicate. Additionally, modal verbs (*e.g.,* should be, must

```
The input_arr is not a null_value.
The input_arr is sorted_in_ascending_order.
```

Fig. 4. Preprocessed specifications listed in Fig. 1.

be, etc.) are replaced with the simple verb ‘be’ in the simple present tense (‘is’) because we treat the sentences as behavioural descriptions of methods. For instance, the statement ‘The method should not return null.’ alone can be understood as there should be some reasons that the method returns null. However, the reason needs to be provided. The proposed approach accepts such a statement without providing the reasons to support the above usage of modal verbs. Otherwise, users can complete the above statement, for instance, ‘The method should not return null if the parameter is not null.’.

Moreover, Java literals (*e.g.,* true, false, null) are preprocessed with a template of ‘a [literal]_value’, where ‘[literal]’ is the original Java literal. The preprocessed phrase complements the NLP to recognise that such a phrase is a noun. Specifications are broken down into sentences using NLTK suite (Bird et al., 2009). Preconditions and postconditions (see Fig. 1 at line 3) is processed according to the above rules and gathered into a separate yaml file. SI annotations, method signature information, parameters and class name are formatted into a program-specific SI library with the same format as the standard SI library. After preprocessing, the specifications are shown in Fig. 4.

4.1.2. Deriving preprocessed sentences with ccg2lambda

The derivation consists of 3 steps: tokenisation, C&C compilation and semantic composition.

1. **Tokenisation**: We apply the tokenisation step from ccg2lambda to the sentences. Tokens are produced by using white spaces as delimiters to split the sentences.
2. **C&C Compilation**: The C&C parser (Curran et al., 2007) from the ccg2lambda pipeline using the enclosed models is invoked. POS tagging is performed, and the syntactic categories of words in the sentences are deduced. Words are lemmatised to their basic forms. Then, the sentences are parsed into CCG trees.
3. **Semantic Composition**: We invoke the semantic composition step from the ccg2lambda pipeline with compositional rules and the semantic template enclosed in the tool, which consists of semantics written in NLTK (Bird et al., 2009) lambda calculus and syntactic category. Once CCG leaves, and nodes are applied using the syntactic category matched template, corresponding semantics are applied to the words, and then β-reductions are applied to generate the MRs. We modified ccg2lambda such that all MR predicates are annotated with their syntactic categories.

The derived MRs of the sample specifications are shown in Fig. 5. Each MR consists of logical operator (*e.g.,* imply, and, etc.) connected quantified formulas, which are constructed by logical operators connected predicates, or quantified formulas. After the preprocessing step, the idioms listed in the library of SI and annotations in the source code are recognised as predicates.

4.2. Translation of meaning representations to JML

For each translation process, three kinds of information are required: (i) MRs of precondition and postcondition; (ii) a standard SI library which consists of reusable target JMLs as well as software engineering idioms and single-word terms; (iii) a program-specific SI library consists of contextual information and annotations in the source code. Notice that (iii) is generated from the preprocessing step in the MR derivation pipeline.

```
exists x.(_input_arr{NN}(x) & TrueP & -exists z2.(_null_value{NN}(z2) &
    TrueP & (x = z2)))
exists x.(_input_arr{NN}(x) & TrueP & exists z1.(_sorted_in_ascending_order{
    NN}(z1) & TrueP & (x = z1)))
```

Fig. 5. Derived MR of the preprocessed specifications listed in Fig. 4.

```
formula: terms
terms: terms connective term
     | terms connective '(' term ')'
     | terms connective NEG term
     | terms connective NEG '(' term ')'
     | term
     | NEG term
     | NEG '(' term ')'
term: predicate_term
     | quantified_term
     | grammar_term
     | KEYWORD_TRUEP
     | IDENTIFIER EQUAL IDENTIFIER
connective: AND
          | EQUIV
          | IMPLY
          | OR
grammar_term: TAG '(' arguments ')'
predicate_term: PREDICATE '{' TAG '}' '(' arguments ')'
              | PREDICATE '{' TAG '}' '(' '(' terms ')' ')'
arguments: arguments COMMA argument
         | argument
argument: IDENTIFIER
        | terms
quantified_term: KEYWORD_QUANTIFIER IDENTIFIER '.' '(' terms ')'
```

Fig. 6. An LR(1) grammar specifies the input MR encoded in HOL.

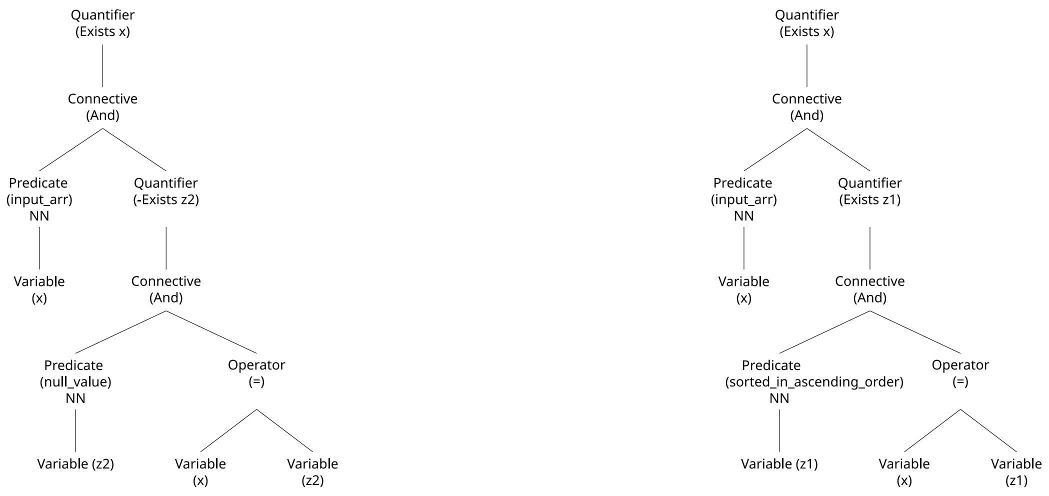

Fig. 7. AST from parsing MRs derived from NLP.

4.2.1. AST construction

The translation begins with building ASTs from MRs to provide data structures for lexical and syntactic analysis. ASTs are produced from parsing MRs with an LR(1) grammar (see Fig. 6) and a lexer.

Each AST node has a token attribute holding the lexical item. There are four types of nodes in the ASTs:

• **Operator** Nodes whose tokens are equal(=). There must be two children for the operator nodes.
• **Connective** Nodes that their tokens are logical operators, such as imply(→), equivalence(↔) as well as and(&).
• **Variable** Nodes that their tokens are the variables. These types of nodes are the leaves of ASTs.
• **Predicate** Nodes that have predicates as their tokens, with children are arguments. Common specifications are written in declarative sentences, which relate subject and object. Therefore, the maximum arity is two. Predicate syntactic categories are stored in the nodes.
• **Quantifier** Nodes of 'Exists' and 'ForAll'.

Predicate node pointers are stored in a queue for direct access in the SI identification. A CST is a table to associate the variables (arguments of the predicates in the HOL) with the symbols that are identified and synthesised during the whole compilation. Each entry in the table is a scope representing a quantified variable (*e.g.,* exist x) in a formula, with such a scope associates all references of the same variable. A CST scope is created if no open scope has the same variable symbol. A variable is associated with a scope when an open CST scope has the same variable symbol. The scope is closed when its variable symbol is resolved in a quantifier rule (*e.g.,* exists x). Renaming variables is achieved using the CST when the same variable symbol is reused. Fig. 7 shows the ASTs of MRs derived from the ccg2lambda pipeline.

4.2.2. Semantics interpretation analysis

After AST construction, the predicate nodes are traversed by using a queue to match their SIs from the standard and program-specific SI libraries (see Algorithm 1). SI of a predicate is matched either (i) a predicate symbol matches a symbol and the syntactic category, or (ii) a predicate with the syntactic category of CD (Cardinal number), which has SI generated in runtime. If a predicate has no SI matched, unless its parent is an 'imply' or an 'equivalent' node or it is the root node, the subtree rooted at this predicate is deleted from the AST. Otherwise, a symbol not found error is thrown to users because the synthesis cannot process without these predicates' SIs being resolved. For instance, "If x then y" is translated to "x implies y" such that "x" and "y" are child nodes to the "imply" node. If "'x" or "y" has no SI, the "imply" cannot be synthesised because JML imply operator is a binary. However, another sentence, "If w x then y" where "w" is an adjective to "'x". The missing of "w" only causes a loss of semantics to the whole sentence. One can consider complementing the SI of "w" if the generated JML for "If x then y" does not satisfy the need. The SI-identified predicates are sorted according to their number of SI arguments in ascending order. Prepositions are exceptions because they can connect complex phrases (*e.g.,* the length of an array). Therefore, they will be synthesised after all other predicates of SI synthesis are finished. All SI-identified predicates' node pointers are stored in a queue.

Algorithm 1 Algorithm of SI Analysis

```
function _SIMATCHER(si, node)
    if $si \rightarrow symbol == node \rightarrow symbol$ and COUNT_ARG($si$) == COUNT_AST_CHILDREN($node$) and $node \rightarrow syntax \in si \rightarrow syntax$ then
        return 0
    else
        return 1
    end if
end function
function SIANALYSIS
    let $silist$ represents a list read from the si libraries
    let $predicates$ represents the queue of all predicates
    let $two\_args\_preds$ represents the queue of predicates that require two arguments
    let $in\_preds$ represents the queue of predicates that categories are preposition
    let $tmp$ represents a temporary queue
    while !ISEMPTY($predicates$) do
        $node$ = DEQUEUE($predicates$)
        if $node \rightarrow syntax == CD$ or (($node \rightarrow syntax == NN$ or $node \rightarrow syntax == Gram\_Rel$)
and $si \rightarrow arg_0 == "*"$) then
            PUSH($tmp, node$)
        else
            $si$ = SEARCHQUEUE($silist, node, \_simatcher$)
            if $si == NULL$ then
                if $node \rightarrow isroot$ or $node \rightarrow parent \rightarrow type != Connective$ or
$node \rightarrow parent \rightarrow symbol == Imply$ or $node \rightarrow parent \rightarrow symbol == Equivalent$ then
                    THROWERROR($Symbol\ not\ found$)
                else
                    DELETE_ASTNODE_AND_EDGE($node$)
                end if
            else
                if $node \rightarrow syntax == IN$ then
                    ENQUEUE($in\_preds, node$)
                else
                    ENQUEUE($two\_args\_preds, node$)
                end if
            end if
        end if
    end while
    while !isEmpty($two\_args\_preds$) do
        enqueue($tmp$, dequeue($two\_args\_preds$))
    end while
    while !isEmpty($in\_preds$) do
        enqueue($tmp$, dequeue($in\_preds$))
    end while
    $predicates = tmp$
end function
```

4.2.3. *Semantics interpretation synthesis*

The proposed SI synthesis process starts with traversing the SI-identified predicates stored in the queue constructed in the SI analysis step. The core concepts of this process are SI alias and subtree SI synthesis.

Algorithm 2 Algorithm of SI alias

```
function SI_alias(node, si)
    $child = node \rightarrow child_0$
    $c =$ get_compiler_symbol_table_pointer($child$)
    remove_children_cst_reference($node$)
    update_cstsymbol_data($c, si \rightarrow interpretation$)
    synchronise_symbol($c$)
    delete_astnode_and_edge($node$)
end function
```

SI alias is a process to synthesise predicates with runtime-generated SI (see Algorithm 2). In this case, there is no SI in the global SI list because the runtime-generated SI is directly stored in the SI pointer of the related nodes. Symbols of the nodes with the variable type and the same variable are replaced by such SI, and the node types are changed to 'Synthesised'. Variable nodes in the same CST scope as the input node are synchronised with the same SI and node type. For instance, a predicate node 'zero' with runtime generated SI '0' accepts one argument '*', and it has one child node with the symbol 'x'. The '0' is stored as the data of the compile-time symbol 'x'. All variable nodes referenced symbol 'x' in the AST are replaced with '0'. The subtree rooted at the predicate node 'zero' is removed from the AST.

Algorithm 3 Algorithm of Subtree SI synthesis

```
function Subtree_SI_Synthesis(node, si)
    $n =$ count_ast_children($node$)
    $s = si \rightarrow interpretation$
    count ← 0
    for all $i = 0 \dots n$ do
        $child = node \rightarrow child_i$
        if $child \rightarrow type == Synthesised$ then
            $s =$ string_replace($s, si \rightarrow arg_i, child \rightarrow symbol$)
        else
            $tmp =$ string_concat($'(', child \rightarrow symbol, ')'$)
            $s =$ string_replace($s, si \rightarrow arg_i, tmp$)
            $si \rightarrow arg_i = child \rightarrow symbol$
            $count{+}{+}$
        end if
    end for
    $node \rightarrow symbol = s$
    if $count == 0$ then
        $node \rightarrow type = Synthesised$
        remove_children_cst_reference($node$)
    else
        $node \rightarrow type = Template$
    end if
end function
```

Subtree SI synthesis is a process performed on a subtree by substituting the leaf nodes' SIs/symbols with the internal node's SI (see Algorithm 3). For instance, a predicate (internal node) 'equal' with matched SI '(x) == (y)' where '(x)' and '(y)' are required arguments, and this predicate node has two child nodes *a* and *b*. The '(x)' and '(y)' in the SI are replaced by the SIs/symbols of *a* and *b*. If any child node is not synthesised, the synthesised result is marked with type 'Template', and the argument is replaced with the variable symbol. This template can be synthesised when the variable symbol is synthesised. Otherwise, all children are synthesised, and the synthesised result is marked with type 'Synthesised'. Regardless of whether the result is marked template or synthesised, the node's symbol is replaced with this result. The child nodes are removed from the AST.

Rules for resolving a predicate are chosen according to their syntax and the number of required arguments in the matched SI. These rules are summarised as the followings:

Rule 1. The SI requires only one argument, which is an asterisk (*): SI alias is applied.

Rule 2. The SI requires only one argument, which is not an asterisk (*): SI synthesis is applied.

Rule 3. The SI requires more than one argument, and all arguments are not asterisks. The predicate node is enqueued if the child node(s) is not synthesised. Otherwise, SI synthesis is applied.

Rule 4. The predicate syntax is an adverb (RB) or preposition (IN), which is the AST's root. This root node is enqueued if it has not been resolved. Otherwise, the node's symbol is replaced by the matched SI.

Rule 5. The predicate syntax is a preposition, or, if one child is not synthesised and is marked with *Multiple SIs*. If the child nodes are not synthesised, the predicate node is enqueued. SI synthesis is applied in the order specified in the SI, such as the reserved keyword _sub that specifies the order of argument substitution (right to left or left to right).

There can be cases that multiple SIs are matched with a predicate. These SIs have the same symbol as the predicate. However, they accept different types of arguments (*e.g.,* Java array and collection types). The types of arguments for synthesis are decided when the arguments are synthesised. Therefore, if a predicate is matched with multiple SIs, it is marked with *Multiple SIs*. It will be resolved after the synthesis of its child (cf. rule 2 and 3) or its sibling (cf. rule 5).

Algorithm 4 Algorithm of operator resolution

```
function opresolution
    let $operators$ represents the queue of nodes with symbols are equal('=')
    while ¬isempty($operators$) do
        $node =$ dequeue($operators$)
        $left = node \rightarrow child_0$
        $right = node \rightarrow child_1$
        $s = NULL$
        if $left \rightarrow type == Template$ and $right \rightarrow type == Template$ then
            throwerror($SI\ conflict$)
        else if $left \rightarrow type == Synthesised$ and $right \rightarrow type == Synthesised$ then
            $s =$ string_concat($left \rightarrow symbol, '==', right \rightarrow symbol$)
        else
            $c = NULL$
            $data = NULL$
            $template = NULL$
            if $left \rightarrow type == Template$ and $right \rightarrow type == Synthesised$ then
                $c = left \rightarrow cst \rightarrow symbol$
                $data = right \rightarrow symbol$
                $template = left \rightarrow symbol$
            else if $right \rightarrow type == Template$ and $left \rightarrow type == Synthesised$ then
                $c = right \rightarrow cst \rightarrow symbol$
                $data = left \rightarrow symbol$
                $template = right \rightarrow symbol$
            end if
            $pattern =$ string_concat($'(', c, ')'$)
            $s =$ string_replace($data, pattern, template$)
        end if
        $node \rightarrow type = Synthesised$
        $node \rightarrow symbol = s$
    end while
end function
```

After all predicate nodes are resolved, the operators queue constructed in the AST construction process is traversed for resolving the subtrees rooted at operator nodes (see Algorithm 4). If both children nodes *a* and *b* have type 'Synthesised', the SI is synthesised as "a == b". If node *a* has type 'Synthesised' and node *b* has type 'Template', we replace the argument in node *b*'s template with node *a*'s synthesised result. The synthesised result from these two cases is stored in the operator node, and all its children are removed. The operator node's type is changed to 'Synthesised'. Both nodes with the type 'Template' indicate a conflict in the SIs. Users can improve their added SIs (in the SI library or annotation) or the natural language comments.

4.2.4. AST simplification

The AST is simplified with the following rules:

- Removal of connective nodes with no children caused by two removed subtrees rooted at the same connective nodes.
- Removal of quantifier nodes because all the variable SIs have already been synthesised.
- Connective and quantifier nodes having only one child are removed, and the child is attached to the node's parent.

Algorithm 5 Algorithm of code generation.

```
function OUTPUT(root)
    let $buffer$ represents a character pointer
    let $stream$ represents allocate a memory stream with the $buffer$
    $haserror = 0$
    WALKTREE($node, stream, \&haserror$)
    if $haserror == 0$ then
        PRINT($buffer$)
    else
        PRINT($Failed$)
    end if
end function
function WALKTREE(node, stream, haserror)
    if $haserror > 0$ then
        return
    else
        if $node == NULL$ then
            $haserror{+}{+}$
            return
        end if
        if $node \rightarrow type != Connective$ then
            PRINTREE($node, stream, haserror$)
        else
            let $operator$ represents a JML operator corresponding to $node \rightarrow conn\_type$
            FPRINTF($stream$, "(")
            WALKTREE($node \rightarrow child_0, stream, haserror$)
            FPRINTF($stream$, "(%s)", $operator$)
            WALKTREE($node \rightarrow child_1, stream, haserror$)
            FPRINTF($stream$, ")")
        end if
    end if
end function
function PRINTREE(node, stream, haserror)
    switch $node \rightarrow type$ do
        case $Synthesised$
            if $node \rightarrow isnegative$ then
                PRINT_TO_FILESTREAM($stream$, "!(%s)", $node \rightarrow symbol$)
            else
                PRINT_TO_FILESTREAM($stream$, "%s", $node \rightarrow symbol$)
            end if
        case $Default$
            THROWERROR($codegenerationfailure$)
            $haserror{+}{+}$
end function
```

4.2.5. Code generation

Finally, after AST simplification, the AST is complete for code generation, through a recursive traversal of the tree (see Algorithm 5). The code generation process starts by calling the *output* method with argument of the AST root node. The *walktree* method is called to perform a recursive traversal of the tree. If the visiting node is null, error is recorded and code generation is failed. If the visiting node is not a connective node, the *printree* method is called directly to generate the target code of the subtree. Otherwise, *walktree* is called to walk the left subtree, followed by resolving the target code of the connective node, and the right subtree's call. The *printree* method prints the input node's symbol (the predicate node's synthesised SI) if the node has already been synthesised. A negative sign is prepended to the node's symbol whenever a node is found negative. All other node types encountered result in code generation failure. After translation, MRs in Fig. 5 is translated to JML shown in Fig. 8.

Table 1
Characterisation of the MRs generated from the program specifications in the benchmark methods. A single MR can contain multiple features, therefore, the number of MRs does not add up to 34.

Characteristics	Number of MRs (out of 34 MRs)
2-level nested quantification	33
3-level nested quantification	6
Nested quantifications as arguments	3
Implications	12
Boolean algebra	13
Propositions	8
Integer arithmetic	6
Reference comparison	4

The aim of using these sample specifications is to provide a demonstration of the proposed approach with simple example. The line of 'requires (!(arr == null));' is translated from 'requires(*The input arr is not a null value.*)', and 'ensures (\forall int k;0 <= k && k < arr.length-1;arr[k]<=arr[k+1]);' is translated from 'The input arr should be sorted in ascending order.'. Readers may already know that JML has a default setting assuming all references are non-null. Therefore, the precondition is true in this case with the postcondition is needed to be checked.

5. Testing and evaluation

To experimentally evaluate the proposed approach, synthesis accuracy and verification accuracy metrics are considered. Synthesis accuracy measures the proportion of synthesised specifications that are correct regarding the syntax and semantics of JML. By design, our compiler fails by providing no output, rather than generating invalid JML. Verification accuracy measures the proportion of valid JML specifications that can be formally checked by a theorem prover against the source code of the specified program. Verification correctness is checked by using OpenJML (Cok, 2011) with the `-esc` flag to enable static checking with z3 on the selected programs with the correctly synthesised JML. The two metrics are computed as follows:

$$accuracy_{synthesis} = \frac{output_{correct}}{output_{correct} + output_{nooutput} + output_{incorrect}} \tag{1}$$

$$accuracy_{verification} = \frac{output_{accepted}}{output_{accepted} + output_{rejected}} \tag{2}$$

Eq. (1) computes number of correctly synthesised JML requirements over the number of programs and Eq. (2) computes the number of correctly verified JML requirements over the number of correctly synthesised JML requirements.

5.1. Experimental setup

Three representative sets of programs were selected for analysis. These benchmark programs are representative of all JML features that are supported by the compiler, *i.e.*, quantification, implication, boolean algebra, propositions, integer arithmetic and pointer comparison. The first set includes five miscellaneous programs that are commonly found in textbooks. The second set includes 53 methods selected from the java.util.Arrays class. After filtering out methods which have identical behaviour (*i.e.*, sorting methods with return types integer and long), eight unique and representative methods are left for consideration. The third set includes eight methods selected from the java.util.Collection class.

In total, there are 21 unique and representative methods in these three sets of programs. In 7 methods, minor modifications to the comments were made to acquire correct NLP, namely implicit subjects, implicit relative pronouns and pronoun references. Natural language sometimes includes implicit information and one may expect future advancements in NLP to overcome these complex language constructs.

```
requires (!(arr==null));
ensures (\forall int k;0 <= k && k < arr.length-1;arr[k]<=arr[k+1]);
```

Fig. 8. Generated JML requirements from the informal specifications in Fig. 1.

```
//@ ensures (*The input arr should be sorted in ascending order.*);
public static void sort(int[] arr){
  ...
}
//@ requires (*The input arr should be sorted in ascending order.*);
public static void binarySearch(int[] arr, int key){
  ...
}
```

Fig. 9. Example showing a sentence can be precondition and postcondition.

The 21 benchmark methods include a total of 34 sentences (distinct MRs) covering the distinct JML features supported.

Table 1 characterises the inputs of the experiment. Each sentence is translated to an MR by ccg2lambda (Martínez-Gómez et al., 2016). All MRs encoded in HOL contain complex structures, for instance, multiple quantifications and implications within a single formula, as well as nested quantifications. The program sets are therefore representative of the complexity and richness of specifications found in real data. Table 2 presents SIs employed in the experiment. These SIs, as per the authors' understanding, represent general expressions applicable to the specified Java types outlined in the provided terms. For instance, the term 'contain' is used to indicate the verification of value existence within a sequence. To address this, either a universal or existential quantified expression proves to be both efficient and comprehensive enough to evaluate all potential values within the sequence. Consequently, the experiments conducted in this study utilise the proposed approach, employing these SIs to assess the program sets. The experiment environment is Ubuntu version 20.04, OpenJML version 0.17-alpha-15, and z3 theorem prover version 4.7.1.

6. Results and discussion

Synthesis of JML is correct in 32 out of 34 program specifications, and the synthesis accuracy is 94% (see Table 3). The two incorrect cases are due to the inability of ccg2lambda to derive correct MRs, in spite of the software engineering idioms added by our approach. In the specification of java.util.Arrays.copyOf (*e.g.,* the original and the copy will have identical values for all valid indices), a noun phrase (identical values) is used as an object, and these phrases should be interpreted as the same predicate for the compound subject (the original and the copy). Nevertheless, such an object is identified as two predicates relating to the two nouns in the compound subject. Such semantics contradict the meaning of "identical". Another failed case is the java.util.Collection.containsAll. Before applying the software engineering idioms knowledge, there is a quantified variable unrelated to any predicate. After applying the knowledge, all variables are related to predicates. However, the object is incorrectly identified by the NLP tool. Improvements on NLP are required in order to produce correct MR in this case, for instance, increasing the size of corpus related to software engineering in training the statistical model used in the C&C parser, and improving the semantic template used in ccg2lambda.

In the java.util.Arrays.asList and the java.util.Arrays.Equals case, the specification uses both 'size' and 'length' to access the size of the array and collection. We have entries of SI in the library with symbols of 'length' and 'size' corresponding to these two types (array and collection). The correctly translated JMLs indicate that the proposed approach can resolve the Ambiguity 1 and Ambiguity 2. Besides, the selected program sets contain complex MR structures, such as implications, nested quantifications, and variables accepted to multiple predicates with different syntax. We have introduced rules that manipulate the synthesis order and process for resolving the predicates. The result shows that the rules are promising in solving the Ambiguity 3. Furthermore, Java literals are mentioned in 16 program specifications. All NLP-derived MRs are correctly translated with the literals tagged as nouns. This result shows that the proposed preprocessing method is promising in solving Ambiguity 4.

Regarding all program specifications for which JML is correctly generated, they are all included in the verification accuracy measure. 31 out of 32 correctly synthesised program specifications can be proved correct by the theorem prover, and the verification accuracy is 97% (see Table 4). In the cases of "sort" and "isPrime", loops implementation is required in the proof. The SMT solver cannot automatically find the proof unless extra invariants and assertions are added to their source code. In the case of java.util.Arrays.deepEquals, recursion proof in traversing the variable depth of multi-dimension array is required. The need to modelling flow variables is out of the scope of this paper. This suggests future work of generating formal specifications for programs with workflows. The above observations support that the proposed approach can bridge the gap between NLP and formal methods, allowing correctly generated MRs to be translated to checkable specifications that can enable deductive verification.

The resulting synthesis accuracy of 94% supports the hypothesis that *it is feasible to improve NLP-based construction of MRs using a knowledge base of programming language-specific idioms and SIs of predicates*. The proposed design achieves a verification accuracy of 97% which supports that *it is possible to design a compiler to synthesise JML specifications from MRs, by resolving ambiguities in higher-order MRs through a combination of symbolic information from the program and the NLP process*.

7. Limitations and threats to validity

The proposed approach requires the explicit informal specifications of preconditions and postconditions. These statements can be referred to as premises and conclusions because premises are logical arguments that support the corresponding conclusions. However, advanced methods still need to be developed to extract these statements from text. The most relevant field is sentiment analysis, known as natural language inference, which determines whether the given hypotheses logically follow the premises.

From this perspective, let us consider the example in Fig. 9. The sentence "The input arr should be sorted in descending order" can serve as a precondition in binary search method and a postcondition in a sorting method. In the given example, the sentence is considered as a postcondition due to presence of the keyword 'ensures'. The complexity of this task for humans highlights the need for advanced NLP techniques to assist in classifying and extracting pre/postcondition information from natural language statements. Consequently, including explicit indications in Javadocs about the intended role of sentences as pre/postconditions can serve as a valuable hint for developers. The mechanism of semantic annotation is intentionally manual, such that developers can quickly complement the semantic interpretation library. Indeed there are possibilities of duplication, currently, the first existing

Table 2
Semantic interpretations used in the experiment.

Term	Syntactic categories	Semantic interpretation	Argument types
sorted in ascending order	JJ	\forall int i;(x);((y))[i-1]<=((y))[i]	Expression, Array
sorted in ascending order	NN	\forall int k;0<=k<(x).length-1;(x)[k]<=(x)[k + 1]	Array
sorted in ascending numerical order	NN	\forall int k;0<=k<(x).length-1;(x)[k]<=(x)[k + 1]	Array
sorted in descending order	NN	\forall int k;0<=k<(x).length-1;(x)[k]>=(x)[k + 1]	Array
greater than or equal to	JJ, JJR, VBG	(x)>=(y)	Primitive, Primitive
less than or equal to	JJ, JJR, VBG	(x)<=(y)	Primitive, Primitive
correspondingly equal to	VBG	(x)!=null && (y)!=null && (x).length==(y).length && ((\forall int i;0<=i <(x).length;((x)[i] == null && (y)[i] == null) ‖ ((x)[i].equals((y)[i]))))	Array, Array
deeply equals to	VBZ	((((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])))))))	Array, Array
length	NN	(x).length	Array
equals to	JJ, JJR, VBG, VBZ	(x) == (y)	Primitive, Primitive
equal to	VBG	(x).size() == (y).size() && (\forall int j; 0<=j<(x).size(); ((x).get(j) == null <==> (y).get(j) == null) ‖ ((x).get(j) != null && (y).get(j) != null <==> (x).get(j).equals((y).get(j)))) && (x).equals((y)) && (y).equals((x))	Collection, Collection
equal to	JJ, JJR, VBG, VBZ	(x) == (y)	Primitive, Primitive
true_value	NN	true	–
false_value	NN	false	–
null_value	NN	null	–
be	VBZ	(x) == (y)	–
return value	NN	\result	–
contain	VBZ	((y)==null && (\exists int i; 0<=i<(x).size(); (x).get(i)==null)) ‖ (\exists int i;0<=i<(x).size();(x).get(i)==(y))	Collection, Primitive
contain	VBD	\exists int i;0<=i< \old((x)).size(); \old((x)).get(i)==(y)	Collection, Primitive
contain	VBZ,VB	(\exists int i;0<=i<(x).length;(x)[i] == (y))	Array, Primitive
removed from	VBG	!(\exists int i;0<=i<(y).size();(y).get(i) == (x))	Collection, Primitive
result	NN	\result	–
even	RB	(x)%2==0	Primitive
prime number	NN	(x)==2 ‖ ((x)> 2 && (\forall int k; (x)> 2 && 2<=k && k<=(x)/2;(x)%k != 0))	Primitive
length	NN	(x).size()	Collection
size	NN	(x).size()	Collection
index	NN	\old(Arrays.asList((x)).indexOf((y)))	Array
if	IN	==>	–
element	NNS	(x)[i]	Array
every element	NN	\forall int i; 0 <= i < (x).length; (x)[i]	Array
reference	NN	(x)	–
change	VBN,VBD	((x).size() != \old((x)).size()) ‖ (\exists int i; 0 <= i < (x).size(); (x).get(i) != \old((x)).get(i))	Collection
empty	JJ	(x).size() == 0	Collection
number of elements	NNS	(x).size()	Collection
for	NNS	\sub(x)2(y)	–
all of the elements	NNS	\forall int i; 0 <= i < (x).length;	Array
all valid indices	NNS	\forall int i; 0 <= i < (x).length;	Array
elements of	JJ	(x)	–
insertion point	NN	(\forall int j; 0 <= j < ((z)); (x)[j] < (y)) && (\forall int j; ((z)) <= j < (x).length; (y) < (x)[j])	Array, Primitive, Primitive
Rel	Rel	_gsub(y)2(x)	–
of	IN	_sub(y)2(x)	–
in	IN	_sub(y)2(x)	–

instance is used. The future improvements such as detecting duplication can be performed.

The poorly written comments become problematic whenever the parser is unable to construct a HOL MR. For instance, regarding to the

Table 3
Synthesis results for the three program sets, with number of methods and specifications, and the number of correctly and incorrectly synthesis that includes incorrect JML as well as those that have no output. The synthesis accuracy is computed using the formula from Eq. (1).

Program sets	No. of unique methods	No. of program specifications	No. of JML correctly synthesised	No. of Inaccurate synthesis	
				No output	Incorrect JML
miscellaneous programs	5	10	10	0	0
java.util.Arrays	8	15	14	1	0
java.util.Collection	8	9	8	1	0
Total	21	34	32	2	0
Synthesis accuracy			$\frac{32}{32+2+0} = 94\%$		

Table 4
Verification results of using correctly synthesised JML with the methods from the program sets. The matrix includes the number of specifications that are correctly synthesised, the number of these synthesised specifications that are accepted and rejected by the theorem prover. The verification accuracy is computed using the formula from Eq. (2).

Program sets	No. of JML correctly synthesised	No. of JML verification accepted	No. of JML verification rejected
miscellaneous programs	10	10	0
java.util.Arrays	14	13	1
java.util.Collection	8	8	0
Total	32	31	1
Verification accuracy		$\frac{31}{31+1} = 97\%$	

```
//@ ensures (*If the username, password, and other relevant information
    are provided, the result will be true.*);
public static boolean login(String username,String password,String token){
  ...
}
```

Fig. 10. Example showing badly rewritten specifications.

poorly written comment listed in Fig. 10, one can follow the procedure below:

1. The term 'other relevant information' is implicitly referred to the parameter token. This can be resolved by adding an annotation '//@ semantics "other relevant information", [NN], 1, [*], token'.
2. The meaning of the verb 'provide' has to be defined. An SI for the word 'provide' can be added to the semantic interpretation library, and the SI can be '(x) != null'.
3. Replacing 'will be' with 'is' can save user's time on writing the SI for the combination of 'will' and 'be', and 'will be' has the same meaning as 'is' regarding to post state of method execution. A preprocessing rule for this replacement has been already added.

The current implementation is based on AST operation rather than HOL beta-reduction. HOL beta-reduction is a syntax level operation, while the substitutions performed for semantic interpretation are semantics level operations. Let S be a sentence, after parsing S, a CCG derivation tree for S is produced, such tree consists of nodes, each of them with a syntactic category. Whenever you have a word/term which is a noun, it can have several different meanings (cf. Section 3). The semantics of a word can only be decided after syntactic analysis and semantic analysis. Therefore, the proposed approach is based on substitutions, which are supported by operations in the AST.

Modifications are necessary to transform the original comments into behavioural descriptions to ensure their usefulness in formal verification. Code comments obtained from reliable sources, such as the official Java documentation, typically explain what a method does, rather than detailing the behaviour of programs before and after the method's execution. Furthermore, advancements in NLP are required to improve the inference of implicit subjects, pronouns, and referencing symbols. These modifications are essential for enabling NLP to generate accurate MRs. It is advisable for code comments to adopt well-structured sentences and explicit components, focusing on the perspective of method behaviour. Consequently, the proposed approach not only facilitates their translation into JMLs but also assists developers in writing code comments that describe the pre/post states of methods.

The JMLs generated and utilised in formal verification are decoupled from the source code. The proposed approach primarily focuses on extracting symbolic information from the source code, encompassing class, method, and parameter identifiers, as well as method return types and parameter types. During the compilation process, this information undergoes rigorous checks for correctness, including type checking, static name resolution, and dataflow analysis. If any errors are detected in the symbolic information, the compiler promptly halts the process and generates specific error messages. As a result, the JML derived from this erroneous information is not employed. Conversely, when the symbolic information is accurately compiled, formal verification proceeds to validate the program's correctness using the JMLs derived from this compilation-checked symbolic information. Therefore, errors in the symbolic information do not propagate to the JMLs utilised for formal verification, ensuring that these JMLs serve as reliable formal specifications for verifying program correctness.

The currently supported JML features in formal verification serve as a representative sample for common programs. These features include requires and ensures clauses with nested quantifications, boolean algebra, reference comparison, propositions, implications, and arithmetic operations. Establishing the current feature set necessitates efforts in observing and understanding the common terms used in software engineering. However, in order to provide support for all desired features, additional programs and specifications need to be incorporated. Our future work aims to expand the range of supported features, striving to accommodate a broader set of capabilities.

8. Related work

The main advantages of our approach are the separation of concerns from NLP and using the contextual information from source code to build a compiler that resolves ambiguity in MRs to generate formal specifications. The closest related work is the ARSENAL framework and methodology (Ghosh et al., 2014), which also aims to translate natural language into formal specifications. Compared to our work, instead of JML specifications, ARSENAL generates SAL models.

In comparing the capability of the generated formal specification, our work is open to generating propositions, Boolean Operations and implications, as well as quantification generation (*e.g.,* forall, exists). We allow the idioms to be specified by developers, whereas ARSENAL determines the idioms by using NLP. To resolve the ambiguity between source code symbols and common words, our work uses the contextual information from source code and the extensible SI library, whereas ARSENAL uses predefined predicates. Besides, our work supports SI annotations that are flexible to developers when they consider SI for specific domains where the SIs in the library are unsuitable. If the same predicate symbols exist in both annotation and SI library with the exact syntactic category, the SI of the annotation is prior to use in the synthesis phase.

Our work focuses on a compiler translating MRs (HOL generated from state-of-the-art NLP) to NLP. In contrast, ARSENAL generates SAL models based on an intermediate logical formula (linear temporal logic). The input complexity is different because linear temporal logic does not have quantifications. Our work additionally provides deductive verification using the generated formal specifications on the programs. Another work (Bertram et al., 2022) has a similar concept in the approach design. This work proposes training a neural model, and ours focus on a symbolic approach. Our work focuses on translating MR to JML, while the neural model is for formulating natural language requirements that are not ready for formal methods.

Compared to solving ambiguities, ARSENAL provides keywords for operator expression (*e.g.,* GREATER THAN) close to our idiom design. However, the keyword design expects users to provide relatively good quality requirements, such as conditional sentences with 'if' and 'then'. Our design with idioms allows users to use non-restricted natural language to specify their programs. In addition, our work evaluates the translated JML using a state-of-the-art deductive verification tool. There is work (Bajwa et al., 2012) aims to resolve syntactic ambiguity caused by poor quality requirements. Our work aims to resolve semantic ambiguities in MRs independent of the original specification quality. Ref. Blasi et al. (2018) suggests using formatted requirement framework such as javadoc to complement the sentence structures. We complement this aspect in the preprocessing step, which allows developers to specify their programs without restrictions in the natural language structure.

Other authors have devised approaches based on statistical models (Zhai et al., 2020; Tan et al., 2007), NLP techniques (Pandita et al., 2012; Goffi et al., 2016), internal defined rules (Blasi et al., 2018), restricting set of natural language mixing with structured model (Bajwa et al., 2010; Motwani and Brun, 2019), analysis and heuristics (Tan et al., 2011, 2012), as well as mixing model and heuristics (Zhou et al., 2017). Ref. Menghi et al. (2019) translates requirements written in a new first-order logic extension language to test oracles specified in Simulink based on predefined rules. Some works (Cabral and Sampaio, 2008; Carvalho et al., 2015) propose approaches using controlled natural language with fixed grammar to generate formal specifications for test case generation. Compared to these approaches, our work focuses on a compiler translating MRs, generated from state-of-the-art NLP, to formal specifications, with knowledge based on idioms, SIs and contextual information from source code and annotations which complement NLP-based MR construction.

In addition, the generated specifications are verified by OpenJML (Cok, 2011), which provides a JML compiler for syntax and semantic checking. Focusing on the application of generating specifications, there are approaches focused on generating contracts from natural language comments for interrupt-related concurrency defects (Tan et al., 2011), lock sensitive defects (Tan et al., 2007), improving test cases generations and testing (Blasi et al., 2018; Goffi et al., 2016; Tan et al., 2012). Our work focuses on applying generated specifications in deductive verification.

There are approaches (Zhou et al., 2017; Ghosh et al., 2014; Cabot et al., 2022) generate specifications or formal models from natural language that can be used in deductive verification, however, none of them is in JML and the scope of contracts (preconditions, postconditions, etc.) is not the same. Ref. Pandita et al. (2012) builds an NLP tool similar to our approach with high translation accuracy, however, the generated code contracts are not in JML format, and they are not able to apply deductive verification. We apply generated preconditions and postconditions to theorem proving with the z3 (de Moura and Bjørner, 2008b), used by OpenJML as the solver, which deductively verifies software against the generated JML specifications, and further validates the consistency of the specification.

9. Conclusion

Meaning representations rely on formal notation and logical structures to capture the meaning of language input, making them a crucial element of natural language processing. However, MRs remain inherently ambiguous in specifying programs when compared to formal specification languages. Therefore, directly translating them into such languages is impractical, limiting their use in the rigorous analysis of software artefacts.

This paper examines the possibility of enhancing NLP-based MR construction with an extensible knowledge base of programming language-specific idioms – combinations of words forming specific meanings – and SIs — fragments in JML that give meaning to the predicates appearing in MRs. This information, combined with the symbols existing in the specified programs, is found to be sufficient to resolve ambiguities in MRs. A compiler to synthesise JML specifications on the basis of such knowledge base is designed and implemented. The compiler is evaluated on a set of representative Java methods, which is limited to 21 methods that nevertheless cover all relevant JML features. Furthermore, the SI library developed for the evaluation only contains interpretations related to the sentences in the dataset. The terms in this library can be generally found in natural language requirements, and hence can be reusable and are not specifically associated to the evaluation dataset.

Results of experimental evaluation on the set of Java methods show that the compiler accurately translates and synthesises correct JML in 94% of the cases. The synthesised JML specifications were statically checked using OpenJML and the z3 theorem prover to evaluate the overall verification accuracy. The static checker is able to prove correctness for 97% of the successful translated cases. Consequently, these results substantiate that the proposed approach contributes to bridging the gap between NLP and formal methods. The main implication for practice is the feasibility of writing comments in natural language that are automatically compiled into a formal specification language. Future work is to refine the compiler to handle more diverse linguistic contexts, deeper insights of the MRs are required by incorporating a broader range of data.

CRediT authorship contribution statement

Iat Tou Leong: Conceptualization, Methodology, Software, Validation, Data curation, Investigation, Writing. **Raul Barbosa:** Conceptualization, Methodology, Supervision, Writing – review.

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.

Data availability

Code and data are made publicly accessible via project MEARC on GitHub.

Acknowledgment

This work is funded by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020.

References

Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., et al., 2005. The KeY tool: Integrating object oriented design and formal verification. Softw. Syst. Model. 4, 32–54.

Bajwa, I.S., Bordbar, B., Lee, M.G., 2010. OCL constraints generation from natural language specification. In: 2010 14th IEEE International Enterprise Distributed Object Computing Conference. pp. 204–213.

Bajwa, I.S., Lee, M., Bordbar, B., 2012. Resolving syntactic ambiguities in natural language specification of constraints. In: Gelbukh, A. (Ed.), Computational Linguistics and Intelligent Text Processing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 178–187.

Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Koehn, P., Palmer, M., Schneider, N., 2013. Abstract meaning representation for sembanking. In: Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse. Association for Computational Linguistics, Sofia, Bulgaria, pp. 178–186.

Barbosa, H., Reynolds, A., El Ouraoui, D., Tinelli, C., Barrett, C., 2019. Extending SMT solvers to higher-order logic. In: Automated Deduction–CADE 27: 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings 27. Springer, pp. 35–54.

Bertram, V., Boß, M., Kusmenko, E., Nachmann, I.H., Rumpe, B., Trotta, D., Wachtmeister, L., 2022. Neural language models and few shot learning for systematic requirements processing in MDSE. In: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering. pp. 260–265.

Bird, S., Klein, E., Loper, E., 2009. Natural Language Processing with Python, first ed. O'Reilly Media, Inc.

Blasi, A., Goffi, A., Kuznetsov, K., Gorla, A., Ernst, M.D., Pezzè, M., Castellanos, S.D., 2018. Translating code comments to procedure specifications. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2018, Association for Computing Machinery, New York, NY, USA, pp. 242–253.

Bos, J., 2016. Squib: Expressive power of abstract meaning representations. Comput. Linguist. 42 (3), 527–535.

Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E., 2005. An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. 7 (3), 212–232.

Cabot, J., Delgado, D., Burgueño, L., 2022. Combining OCL and natural language: A call for a community effort. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. MODELS '22, Association for Computing Machinery, New York, NY, USA, pp. 908–912.

Cabral, G., Sampaio, A., 2008. Formal specification generation from requirement documents. Electron. Notes Theor. Comput. Sci. 195, 171–188, Proceedings of the Brazilian Symposium on Formal Methods (SBMF 2006).

Carvalho, G., Barros, F., Carvalho, A., Cavalcanti, A., Mota, A., Sampaio, A., 2015. NAT2test tool: From natural language requirements to test cases based on CSP. In: Calinescu, R., Rumpe, B. (Eds.), Software Engineering and Formal Methods. Springer International Publishing, Cham, pp. 283–290.

Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E., 2005. Beyond assertions: Advanced specification and verification with JML and ESC/JAVA2. In: International Symposium on Formal Methods for Components and Objects. Springer, pp. 342–363.

Cok, D.R., 2011. OpenJML: JML for java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (Eds.), NASA Formal Methods. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 472–479.

Cok, D.R., Kiniry, J.R., 2004. Esc/java2: Uniting ESC/JAVA and JML. In: International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. Springer, pp. 108–128.

Curran, J., Clark, S., Bos, J., 2007. Linguistically motivated large-scale NLP with C&C and boxer. In: Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions. Association for Computational Linguistics, Prague, Czech Republic, pp. 33–36.

de Moura, L., Bjørner, N., 2008a. Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340.

de Moura, L., Bjørner, N., 2008b. Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340.

Delahaye, M., Kosmatov, N., Signoles, J., 2013. Common specification language for static and dynamic analysis of C programs. In: Shin, S.Y., Maldonado, J.C. (Eds.), SAC 2013 - 28th ACM Symposium on Applied Computing. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013), vol. 2, ACM, Polytechnic Institute of Coimbra (IPC), Coimbra, Portugal, pp. 1230–1235, Session: Volume II: Software development and system software & security: software verification and testing track.

Filliâtre, J.-C., Paskevich, A., 2013. Why3—where programs meet provers. In: Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22. Springer, pp. 125–128.

Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W., 2014. Automatically extracting requirements specifications from natural language. CoRR abs/1403.3142, arXiv:1403.3142.

Goffi, A., Gorla, A., Ernst, M.D., Pezzè, M., 2016. Automatic generation of oracles for exceptional behaviors. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. In: ISSTA 2016, Association for Computing Machinery, New York, NY, USA, pp. 213–224.

Jones, C.B., 1986. Systematic Software Development Using VDM. In: International Series in Computer Science, Englewood Cliffs, N.J.

Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B., 2015. Frama-C: A software analysis perspective. Formal Aspects Comput. 27, 573–609.

Leavens, G.T., Baker, A.L., Ruby, C., 2006. Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes 31 (3), 1–38.

Leino, K.R.M., 2010. Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 348–370.

Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J.R., Bethard, S., McClosky, D., 2014. The stanford CoreNLP natural language processing toolkit. In: Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations. pp. 55–60.

Martínez-Gómez, P., Mineshima, K., Miyao, Y., Bekki, D., 2016. ccg2lambda: A compositional semantics system. In: Proceedings of ACL 2016 System Demonstrations. Association for Computational Linguistics, Berlin, Germany, pp. 85–90.

Menghi, C., Nejati, S., Gaaloul, K., Briand, L.C., 2019. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, pp. 27–38.

Meyer, B., 1997. Object-Oriented Software Construction, vol. 2, Prentice hall Englewood Cliffs.

Motwani, M., Brun, Y., 2019. Automatically generating precise oracles from structured natural language specifications. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, pp. 188–199.

Owre, S., Rushby, J.M., Shankar, N., 1992. PVS: A prototype verification system. In: International Conference on Automated Deduction. Springer, pp. 748–752.

Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S., Paradkar, A., 2012. Inferring method specifications from natural language API descriptions. In: 2012 34th International Conference on Software Engineering. ICSE, pp. 815–825.

Schmitt, X., Kubler, S., Robert, J., Papadakis, M., LeTraon, Y., 2019. A replicable comparison study of NER software: StanfordNLP, NLTK, OpenNLP, SpaCy, Gate. In: 2019 Sixth International Conference on Social Networks Analysis, Management and Security. SNAMS, IEEE, pp. 338–343.

Slind, K., Norrish, M., 2008. A brief overview of HOL4. In: International Conference on Theorem Proving in Higher Order Logics. Springer, pp. 28–32.

Steedman, M., 1996. Surface structure and interpretation. In: Linguistic Inquiry.

Tan, S.H., Marinov, D., Tan, L., Leavens, G.T., 2012. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. pp. 260–269.

Tan, L., Yuan, D., Krishna, G., Zhou, Y., 2007. /*icomment: Bugs or bad comments?*/ In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles. SOSP '07, Association for Computing Machinery, New York, NY, USA, pp. 145–158.

Tan, L., Zhou, Y., Padioleau, Y., 2011. Acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, Association for Computing Machinery, New York, NY, USA, pp. 11–20.

Warmer, J.B., Kleppe, A.G., 2003. The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional.

Zhai, J., Shi, Y., Pan, M., Zhou, G., Liu, Y., Fang, C., Ma, S., Tan, L., Zhang, X., 2020. C2S: Translating natural language comments to formal program specifications. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, pp. 25–37.

Zhou, Y., Gu, R., Chen, T., Huang, Z., Panichella, S., Gall, H., 2017. Analyzing APIs documentation and code to detect directive defects. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, pp. 27–37.

Iat Tou Ieong is a Ph.D. student at the University of Coimbra. He received the Master degree in computer science from National Chiao Tung University, Taiwan. At MTel Telecommunication company he was a deputy manager for value-added service development. At the Macao Health Bureau he is a security officer and development team lead responsible for the electronic health record project. His research interest focus on using natural language processing techniques and formal verification in software engineering development to achieve software reliability.

Raul Barbosa is a Tenured Assistant Professor at the University of Coimbra. He received the Ph.D. degree in computer engineering from Chalmers University of Technology. At Carnegie Mellon University he was an Adjunct Associate Teaching Professor in the Institute for Software Research. He collaborated and was the principal investigator at UC in diverse research projects. His research interests focus on reliable software and distributed systems, including principles for designing and evaluating computer systems that must ensure safety and availability. His research activity is currently centred on projects addressing cloud computing, dependable software architectures and reliable AI. These topics are systematically addressed using formal approaches such as model checking and experimental approaches such as fault injection. His teaching activities take place at the Department of Informatics Engineering of the University of Coimbra.