C2S: Translating Natural Language Comments to Formal Program Specifications

Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, Xiangyu Zhang

1 INTRODUCTION

Formal specifications are vital for many software engineering tasks. Program verification requires procedure specifications to verify properties of interest [14, 44, 45], especially when the source code is unavailable or too complicated to analyze. Synthesis techniques need specifications to synthesize unknown expressions [10, 24, 38]. Software testing demands specifications to generate test oracles [15, 20, 25, 27]. Program debugging requires specifications to locate root causes [22, 46]. As such, a lot of work has been devoted to designing formal languages for specification composition. Java modeling language (JML) is one of such specification languages and widely used by developers (e.g., to provide specifications for JDK library methods [8]). However, manually composing formal specifications is not only time-consuming and error-prone, but also requires substantial expertise. This motivates us to develop an automatic approach to synthesize program specifications.

Modern software projects have abundant natural language (NL) documentation which provides a wealth of semantic information about code properties and behaviors. For example, in the Linux kernel, FreeBSD, Open-Solaris, MySQL, Firefox, and Eclipse, 21.8–29.7% (0.3–1.7 million lines) of their code bases are code comments [40]. J2SE’s Javadoc [6] is a representative example document. It contains rich information such as properties of parameters and desired behaviors of methods. Such comments are in natural language and describe program semantics informally. Recently, natural language processing (NLP) techniques have achieved enormous success in natural language understanding (NLU) and natural language generation (NLG). However, formal program specifications are essential for various software engineering tasks, such as program verification, program synthesis, code debugging and software testing. However, manually inferring formal program specifications is not only time-consuming but also error-prone. In addition, it requires substantial expertise. Natural language comments contain rich semantics about behaviors of code, making it feasible to infer program specifications from comments. Inspired by this, we develop a tool, named C2S, to automate the specification synthesis task by translating natural language comments into formal program specifications. Our approach firstly constructs alignments between natural language word and specification tokens from existing comments and their corresponding specifications. Then for a given method comment, our approach assembles tokens that are associated with words in the comment from the alignments into specifications guided by specification syntax and the context of the target method. Our tool successfully synthesizes 1,145 specifications for 511 methods of 64 classes in 5 different projects, substantially outperforming the state-of-the-art. The generated specifications are also used to improve a number of software engineering tasks like static taint analysis, which demonstrates the high quality of the specifications.
progress and have been adopted in many software engineering tasks showing fairly promising results [21, 28, 51, 52]. This inspires us to utilize NLP techniques to address the automatic specification generation challenge.

There are existing techniques that generate specifications from code comments, including @tComment [43], Toradocu [20] and Jdoctor [15]. They firstly manually define a set of patterns. Each pattern specifies a template for NL comments and a corresponding template for specifications. Then they match a given NL comment against the patterns to generate specifications. However, such pattern-matching based techniques require many manual efforts and can hardly handle the flexibility and the diversity of natural languages, which results in limited generality. Specifically, @tComment infers null value related properties to detect comment-code inconsistencies. It cannot handle cases related to other properties such as “throws NoSuchElementException if this deque is empty”.

Toradocu generates conditional expressions from exception-related comments to create test oracles for exceptional behaviors. The conditions they support include “something is/are positive/negative/true/false/null/<1/<=0”. Such patterns are still limited and they cannot handle the aforementioned (typical) comments. Based on Toradocu, Jdoctor derives specifications from return-value-related comments in addition to exception conditions, showing very promising results and representing the state-of-the-art. However, since it still relies on patterns, we found that many comments cannot be handled as they are not covered by the patterns, such as “Returns the first element in this list”. In addition, Jdoctor is incapable of generating specifications to describe the main functions of void methods. For example, Jdoctor cannot generate any specification for the void method clear() whose comment is “Removes all of the elements from this set”. Finally, existing work cannot generate specifications to describe normal functional behaviors except return-value-related behaviors, which are prevalent in comments. Hence our goal is to develop a general approach that can automatically synthesize specifications from different kinds of comments.

In this paper, we propose C2S, an automatic approach of translating NL comments of a target method to formal program specifications by assembling primitive tokens guided by the specification syntax and the context of the target method (method properties such as parameters). The primitive tokens are automatically extracted from existing JML specifications [8] written for JDK library methods. Specifically, we regard NL comments and JML specifications as two languages expressing the same semantics, and formulate the specification translation task as a syntax-guided synthesis problem. We automatically couple NL comments and corresponding JML specifications to build a bilingual corpus to construct alignments between NL words and specification tokens. Existing JML specifications contain information specific to their methods. Such information is abstracted away to achieve generality. Then for the target method with comments, we extract the generalized tokens that are associated with the words contained in an NL comment from the alignments and assemble them to synthesize program specification candidates guided by the specification syntax and the concrete context of the method. The aforementioned generalized tokens used in the candidates are substituted with concrete ones for each target method after the synthesis process. Testing is further used to filter out incorrect candidates.

We make the following contributions:

- We propose a novel search-based technique to automatically translate NL comments to formal program specifications that specify the expected set-ups (preconditions) of using a method and the effects of executing a method (postconditions for both exceptional behaviors and normal behaviors). Our approach avoids the manual and error-prone efforts of defining patterns.
- We develop a prototype C2S based on the proposed idea, and evaluate it on 511 methods of 64 classes in 5 different projects. The applications of these specifications in dynamic testing and Android app static taint analysis demonstrate that these specifications precisely represent the method behaviors and improve the efficiency and effectiveness of various analysis and testing applications.

2 MOTIVATION

Instead of using formal program specifications to convey code semantics, developers tend to use natural language comments to informally describe semantics. Fig. 1(k) demonstrates a real-world method whose semantics are explained using three natural language sentences. Line 81 describes that the main functional behavior is to “remove the first element from this list and also return this element”, which is a post-condition of normal behavior. Line 82 points out that “if this list is empty, NoSuchElementException will be thrown” which is a post-condition of exceptional behavior. Line 83 specifies the return value. There are existing efforts in analyzing NL comments to generate formal program specifications. However, no existing work can infer specifications from the three comments in Fig. 1(k). The existing approaches all rely on patterns summarized manually from comments to derive specifications, which requires substantial manual work and the patterns can only work on very limited comments. Moreover, an NL comment can be interpreted differently in different contexts. For example, “the first in “returns the first component” (line 62) in Fig. 1(g) means “the first component in the receiver object before/after executing the method” (the method execution does not change the receiver object) while “the first in “returns the first element” (line 83) in Fig. 1(k) means “the first component in the receiver object before executing the method” (the method execution removes the first element in the receiver object). The existing work generates the same specification for a given pattern without considering contexts, which may induce errors. We showcase how our approach can address these limitations using the examples in Fig. 1.

JML is a program specification language designed to specify desired properties/behaviors of Java classes and methods. Fig. 1(b) shows an excerpt of JML specification for method remove(int) of class java.util.ArrayList. JML can specify both exceptional behaviors (lines 12-13) and normal behaviors (lines 16-17). There are some existing JML specifications which can be associated with corresponding NL comments. This motivates us to design an approach to leverage both NL comments and existing JML specifications to automatically infer specifications from comments. Given that the dataset of existing specifications is minuscule, we resort to the search-based technique rather than machine learning techniques which have the overfitting problem.
Fig. 1(a), Fig. 1(c), Fig. 1(e) and Fig. 1(g) are four documentation samples where NL comments are annotated using different tags like @returns to describe different aspects. Fig. 1(b), Fig. 1(d), Fig. 1(f) and Fig. 1(h) separately show the corresponding JML specifications composed by developers. The specifications are annotated using different tags to specify different kinds of behaviors. For example, a specification annotated by ensures specifies a post-condition which is a property held by a method when the method finishes execution normally. In the first stage of our technique, we automatically couple each specification with the corresponding comment based on annotations. By respectively coupling comments ①, ②, and ③ with specifications ⑤, ④ and ⑥, we get the three comment-specification pairs shown in Fig. 1(i).

Then comments and corresponding specifications are automatically pre-processed and split to obtain alignments between NL words and specification tokens shown in Fig. 1(j). The first column lists NL words and the second column lists all the tokens associated with words in a method comment are used to assemble specifications for the method.

Notice that we have a token p@int in Fig. 1(j), but it does not occur in any JML specification. Here p@int is a parameter placeholder used to substitute the first parameter index with the data type int for method remove(int index). Also, we have method names like get and contains. These are pure methods (i.e. methods that do not have side-effects) used in the collected JML specifications to encapsulate primitive actions. We directly use these methods as our method placeholders. Our system substitutes information specific to existing methods with placeholders to derive generalized representations, making our approach more general and more efficient. We refer to each processed specification as an intermediate...
representation (IR) of the specification. Each IR is an AST which contains the syntax information of the specification. Such syntax information is essential to assembling tokens.

Given a target method with comments (Fig. 1(k)), we first preprocess each comment into a bag of words and then retrieve the potential IR tokens from the word-token pairs (Fig. 1(j)). Considering the comment in line 82, we can retrieve tokens like “isn\textit{isEmpty}” and “\textit{this}” given the NL word “\textit{empty}” and tokens like “\textit{\rightarrow}” given the NL word “\textit{if}”. With these tokens, we leverage grammar rules of specifications to synthesize potential IRs shown in Fig. 1(m). This is feasible since the IRs have a limited number of tokens and syntactic structures (e.g., a method only accepts parameters of fixed types).

Next we instantiate IRs into specification candidates. Take the IR in Fig. 1(o) as an example, the instantiation consists of several steps like replacing the parameter placeholder \texttt{p1@E} (first parameter with generic type \texttt{E}) with the formal parameter name \texttt{v} and substituting the method placeholder \texttt{contains} (\texttt{b} in Fig. 1(o)) with the concrete method name \texttt{contains\texttt{Vertex}} (\texttt{c} in Fig. 1(p)), depending on the context of the target method.

The last step is to filter out incorrect candidates via testing and the result are shown in Fig. 1(n), in which the candidates like \texttt{d} in Fig. 1(m) are filtered out. Notice that we generate two specification candidates \texttt{f} and \texttt{g} for the comment “\texttt{the return value is the first element from this list}” (line 83 in Fig. 1(k)). Specification \texttt{f} specifies “\texttt{the return value is the first element in the receiver after executing the target method}” which is the case for line 62 in Fig. 1(g). Specification \texttt{g} specifies “\texttt{the return value is the first element in the receiver before executing the target method}” which is the case for line 83 in Fig. 1(k). We leverage testing to prune out the wrong one to provide the context-aware generation.

Improving Testing. Normally, it is difficult for automatic testing tools to have a general way to generate test cases to cover diverse non-exceptional behaviors. However, our derived specifications can be used to generate more accurate and more effective tests. Consider the method \texttt{add\texttt{Vertex}(V)} in Fig. 1(l). The test cases generated by Randoop [9] only check whether the return value is true or false, which cannot check the core behavior of adding a vertex. In contrast, we can use our generated specification (box \texttt{f} in Fig. 1(p)) to guide a testing tool to check whether the input \texttt{v} is successfully added. As we will show in Section 4, the generated specifications can be used in generating new test oracles, reducing false alarms in automated testing, and improving static taint analysis.

3 DESIGN

Fig. 2 gives the design of our approach, which includes the search space preparation phase (left) and the synthesis phase (right). The inputs of the first phase are NL comments collected from Java documentation [6] and corresponding JML specifications collected from JML website [8]. We begin by using the association engine to automatically couple each specification with an NL comment and the generated comment-specification pairs are fed into the tokenizer to build word-token pairs which will be used to synthesize specifications in the second phase. For each comment, the pre-processor cleans it and splits it into individual words. For each specification, the IR translator substitutes the concrete subjects (e.g., parameter names) specific to the subject method with placeholders to obtain a general specification, and builds an AST from the generalized specification. After that, NL words are coupled with AST tokens that will be used as the search space for synthesis.

The goal of the second phase is to generate specifications for a target method with NL comments. Firstly, we obtain NL words from a cleaned NL comment using the pre-processor. Then the IR synthesizer generates IR candidates by obtaining AST tokens that are associated with the NL words from the word-token pairs and assembling the tokens into candidates based on grammar rules and the context of the method. After that, the specification generator instantiates each IR candidate with the context of the target method (e.g., parameters of the method) to obtain a formal specification. Specifically, candidates are generated from tokens extracted from existing specifications and thus parameter placeholders and method placeholders used in the candidates are specific to the original data. As such, they need to be separately instantiated with formal parameters of the target method and concrete Java methods in the class/superclass containing the target method. Lastly, the specification checker leverages existing developer test cases to filter out incorrect specifications.

3.1 Specification Language

JML specifications are used to specify properties and behaviors of Java methods. We simplify and formalize the specification language and present the abstract language model in Table 1. Our approach generates both normal specifications and exceptional specifications for a given method. Normal specifications specify preconditions and post-conditions when a method terminates execution without throwing an exception. Exceptional specifications specify a specific exception is thrown under a certain condition. Logical expressions (LE) lists different types of specifications that can be used to describe normal specifications. Throw expression (TE) specifies when a certain conditional expression (CE) is true, a specific exception (EL) will be thrown.

Specifications cannot have side effects on objects, otherwise the program state may be changed. Therefore, expressions like assignments and increments are not allowed in the specification language. Also only methods that have no side-effects on a program state can be used in specifications.

In addition to the logical expressions supported by Java, forall expressions (FE) and implication expressions (IE) are introduced to describe program states after executing a method. FE represents universal quantification expressions. For example, the specification
The association engine automatically couples specifications with program states before calling a method. This enables us to represent a method whose specifications are being synthesized. We will use the specification in line 12 with the comment "if the index is out of range (index < 0 || index ≥ this.size()) an exception is thrown" as an example to show that "the method remove(int) is expected to return the element previously at the position i of the receiver list when the method remove has not been called".

In order to represent the return value of calling a method, the keyword "result" is introduced. The data type of Result depends on the return type of the method. With result, we can have the specification \( \text{result} == \text{old(list.get(i))} \) for the above-mentioned example to convey that "the method remove(int) is expected to return the element previously at the position i of the receiver list before being modified by the method invocation".

3.2 Association Engine

The association engine automatically couples specifications with corresponding comments based on annotations to prepare comment-specification pairs.

In documentation, a method has comments for method parameters, exceptional-behaviors and normal-behaviors. As shown in Fig. 1(a), a parameter comment annotated by @param (e.g., line 04) gives a brief parameter description. In some cases, a parameter comment may describe the condition that the parameter should satisfy in order not to make the method execute exceptionally. For example, method subtract(iterable a, iterable b) of class CollectionUtils in project Apache Commons Collections [1] has a parameter comment "a must not be null" which indicates a precondition. An exceptional-behavior comment annotated by @throws (e.g., line 02) describes the condition that triggers an exception of a specific type.

For normal behaviors, the first sentence (e.g., line 01) of the comments of a method is a concise but complete description of what the method does [5, 23], and the comment annotated by @returns (e.g., line 05) describes the return value of the method.

Similarly, a method has both exceptional-behavior specifications and normal-behavior specifications, which also can be distinguished by their annotations. Take Fig. 1(b) as an example, in exceptional behaviors, the specification index < 0 || index ≥ this.size() annotated by requires (line 12) specifies the condition of throwing exception java.lang.IndexOutOfBoundsException (line 13 annotated by signals_only). Based on the exception type, we can associate the specification in line 12 with the comment "if the index is out of range (index < 0 || index ≤ size())" in line 02 (pair @ in Fig. 1(i)). In normal behaviors, when the precondition annotated by requires in line 15 is met, the method will terminate the execution normally in a program state that satisfies the post-conditions annotated by requires (lines 16-17). The specification in line 16 is an equality expression with one operand as \( \text{result} \) meaning the specification describes the return value. And hence we associate such specifications with comments annotated by @returns (pair @ in Fig. 1(i)). Other type of post-conditions describe execution effects of a method and they are associated with the first sentence that summarizes the method (pair @ in Fig. 1(i)).

3.3 Tokenizer

The tokenizer accepts a comment-specification pair and transforms it into pairs of NL words and AST tokens. For each input pair, the pre-processor is leveraged to clean the NL comment and split it into separate words, and the IR translator is leveraged to convert the specification into an AST. Then the tokenizer constructs pairs of NL words and AST tokens by coupling each word in the cleaned comment with each leaf node in the AST.

3.3.1 Pre-processor

To acquire more general comments by removing unnecessary information and normalizing texts, the pre-processor mainly performs three tasks: 1) Removing stop words (common words appearing frequently [37] like "the"); 2) Reducing derived words to their word stem, namely root form, by applying the Porter stemming algorithm [34]. For example, the word "inserts" is transformed into "insert"; and 3) Lowercasing all the words. After the cleaning, each comment is split into individual words by space.

3.3.2 IR Translator

The IR translator generalizes a JML specification in the text form to an abstract form and parses the abstracted specification to an IR represented using AST.

Each generalized specification is called an IR with the semantics preserved. There are two main reasons for using IRs. The first one is to reduce the search space for the synthesis process and the second one is to facilitate the instantiation of the synthesis results with concrete information belonging to the target method (i.e., the method whose specifications are being synthesized). We will use the specification this.contains(e) of method add(E e) in Fig. 1(d) to demonstrate the reasons as well as the process.

Firstly, we substitute all concrete parameter names with parameter placeholders in the form of \( \text{p} i \) (the i-th parameter with type t). Such substitution enables us to achieve better generality and higher efficiency. For example, assume the parameter e is not generalized and hence becomes part of the token set that would be used to synthesize a specification. However, it is very likely a target method.
Algorithm 1 Synthesizing IR Candidates from a NL Comment

```
1: procedure synthesizeIRs(C, P, T, M)
2:   wordSet ← preprocess(C)
3:   tokenSet ← extractTokens(wordSet, M)
4:   S ← filterTokens(tokenSet, P, T)
5:   while true do
6:     removeSet ← ∅
7:     oldSet ← S
8:     set1 ← S
9:     set2 ← S
10:    for each ast1 ∈ set1 do
11:       for each ast2 ∈ set2 do
12:         S ← S ∪ assemble(ast1, ast2, P, T)
13:       removeSet ← removeSet ∪ ast1
14:     if S == oldSet then
15:        break
16:     S ← S \ removeSet
17:   return filter(S)
```

with similar specifications does not have a parameter e. As such, the generated specification is invalid. It would be very difficult to replace it with some other parameter name as we do not know what e represents. However, with p1@E, we know that it is the first parameter with a generic type and thus finding a replacement is much easier. Another benefit of using placeholders is to reduce the search space by decreasing the number of tokens used to construct specifications since parameter names can be diverse.

Pure methods which do not have side-effects are used in JML specifications to encapsulate primitive actions. For example, the method contains in the specification this.contains(e) is a pure method. We use the pure methods as method placeholders in our IRs and they will be substituted with concrete methods when IRs are instantiated into specifications for a target method (Section 3.5).

After generalization, IR translator parses a specification to an AST. The parser performs lexical analysis and syntax analysis, which is very similar to that in a compiler [11].

ASTs. We represent IRs using ASTs, and use leaf nodes to represent tokens and non-leaf nodes to represent the IR’s grammatical terms. As shown in Fig. 3, leaf nodes like "result" and "==" are tokens used to constitute an expression, and a non-leaf node like "Equality Expression" represents a term having its inner structure.

3.4 IR Synthesizer

In this section, we introduce our search-based approach of synthesizing IRs for a method with NL comments.

The process of synthesizing IRs is presented in Algorithm 1. It takes a method comment C, the parameters with type information P, the return type T, and the word-token pairs M as inputs and the output is a set of IR candidates, denoted as S. Given a method comment, we first obtain the initial token set from word-token pairs M based on words of the pre-processed comment (lines 2-3). Then we leverage the parameter data type(s) P and the return type T of the target method to eliminate some tokens from the initial token set (line 4). For example, if the target method has no parameters, then the token p1@int should be removed from the token set. The result set of tokens is stored in S to be used to synthesize IRs.

With the token set, the main procedure iteratively assembles two ASTs in the set S into a larger AST based on the specification language model shown in Table 1. When a fixed point is reached, we terminate the assembling process. The assembling procedure in line 12 treats the two input ASTs (i.e., ast1 and ast2) as siblings to form a new AST. Note that the assemble procedure only generates an AST with correct syntax. In addition to the syntax rules in Table 1, we also check the syntax based on the following properties of the target method: 1) the number of parameters; 2) the data type of each parameter; and 3) the return type. Suppose that we have two ASTs separately for "result" and "null", the IR result != null is not generated for a target method whose return type is int although it is a correct IR for some methods. Note that two different ASTs may have the same semantic, such as ASTs of result != null and null != result. Such ASTs are regarded as duplicated and they will not exist simultaneously in the set S.

Line 13 checks whether a new AST is produced in the loop (lines 11-12). If not, it is highly possible that no new ASTs can be constructed from ast1 in the following iterations and thus we add it to removeSet and remove it from S in line 17. Consequently, ast1 will not be used to synthesize IRs in later iterations.

After the assembling process, we get an AST forest, which may contain incomplete ASTs (e.g., a relational expression missing one operand) and complete ASTs that do not denote logic assertions (e.g., this.size() whose value is not boolean). We eliminate such ASTs and return the remaining ones as IR candidates (line 18).

Example. Fig. 3(a) presents some IR candidates assembled from the tokens listed in Fig. 3(b). Some non-leaf nodes are omitted for the limitation of space. In the first iteration, ASTs (1) and (2) are constructed. Then based on the initial tokens in Fig. 3(b) and the ASTs created in the first iteration, (3) and (4) are assembled in the second iteration. After that, the third iteration produces (5) and (6). Finally, the whole tree in Fig. 3(a) is built in the fourth iteration. In addition to the whole tree, three other complete logical specifications are synthesized in this process, namely (5), (4) and (6). All the four IRs are returned by the synthesis algorithm and the ones that do not specify the desired behaviors of the method will be eliminated by the specification checker in Section 3.6. Other synthesized IRs like result != 1 && this.contains(p1@Object) are omitted here.

3.5 Specification Generator

The specification generator translates an abstract IR to a concrete specification, leveraging the contextual information of the target
The goal of the specification checker is to eliminate invalid specification candidates. It works by first transforming the specification candidates into test oracles and then adding these oracles to existing project test cases written by developers. Here, we trust these project test cases as they were the ones used in unit testing and regression testing. As such, any specification candidates that trigger violations in these tests are deemed invalid.

The specification checker consists of two main steps. The first step is to instantiate a specification by substituting general information (e.g., formal parameters) with concrete information (e.g., actual parameters). The second step is to instrument existing test cases with assertions generated from instantiated specifications and necessary Java statements.

3.6.1 Specification Instantiation. General information in specifications is instantiated in this step. Recall that our specifications are generated for individual method. These specifications use formal parameters, this and \( \text{\textbackslash result} \) to respectively represent inputs, the receiver and the return value of a method, which has to be instantiated in order to generate test oracles. For a statement (in a project test case) that invokes the target method, the specification checker automatically instantiates them by using values in the method invocation statement. Notice that these concrete values are also used to generate assertion statements for the test case (Section 3.6.2).

This instantiation process is shown in Algorithm 2. It takes a specification \( S \) of the method \( M \), a statement \( ST \) that invokes the method \( M \) as well as the method \( M \) as inputs, and returns the instantiated specification that is specific to the input statement \( ST \). We will use the example in Fig. 4 to show how it works. Fig. 4(a) gives a statement from an existing test case which invokes method \( \text{remove(int index)} \) with two generated specifications listed in Fig. 4(b).

The \( \text{instantateSpecification} \) procedure defined in line 1 recursively calls itself to do substitution. It has three base cases separately for return value, parameters and receiver. When the expression \( \text{\textbackslash result} \) is found (line 3), we call the procedure \( \text{getReturnValue} \) in line 4 to get the actual return value of the given statement \( ST \). For example, the statement in Fig. 4(a) uses variable \( ret \) to store the return value and thus \( ret \) is used to substitute \( \text{\textbackslash result} \) (1 to 3). When one
formal parameter \( p \) of method \( M \) is found (line 5), the procedure `getActualParameter` in line 6 is called to retrieve the corresponding actual parameter. For example, the formal parameter `index` in Fig. 4(b) is substituted with the actual parameter 2 in Fig. 4(c) (1 to 3, and 2 to 4). When the keyword `this` is found (line 7), we substitute it with the actual receiver of method \( M \) returned by the procedure `getReceiver` (line 8). The statement in Fig. 4(a) has `list` as the receiver, and thus this is substituted by `list`.

To express the semantics of `old`, we propose to save the program state (i.e., the state of the receiver in Java) before executing a statement into a new variable \( r \), a clone of the receiver, and later retrieves from \( r \). More specifically, when the expression `old(e)` is found (line 9), we recursively call the procedure `instantiateSpecification` to substitute the aforementioned general information in \( e \) and store the substituted expression to \( e' \) (line 10). Then we substitute the receiver of \( M \) in \( ST \) ( \( r \) in line 11) with our cloned receiver \( r \) (line 12). Consequently, we can retrieve the program state before executing \( ST \) from \( r \). For example, in Fig. 4(d), we can get the third element in the original linked list (i.e., before calling `remove(int)` in line 02) from receiver `oldList` cloned in line 01. The previous third element cannot be obtained from the receiver `list` in list 03 since it has been removed by calling `remove(int)` in line 02.

For other expression types, each operand is substituted by recursively calling the procedure `instantiateSpecification` to get the whole expression instantiated (lines 13–18).

3.6.2 Assertion Instrumentation. After deriving instantiated specifications, the specification checker instruments test cases with assertions (and related code). The rules of instrumenting assertions and code into existing test cases are shown in Fig. 5. The instrumentation process works recursively and is named as `generateOracle`. The first column lists the instantiated specifications and the transformed test cases are boxed in the third column. The statement `st` here calls a target method for which we have synthesized specifications although normally the negation of the exception triggering condition could be considered as essentially a precondition.

To answer RQ1, we measure precision and recall of C2S with 5 frequently-used Java libraries with well-maintained documentation, namely JDK [6], Apache Commons Collections [1], Guava [4], GraphStream [3], and JGraphT [7]. The results are shown in Table 2, which presents the project (column 1), the number of classes/methods (columns 2/3) that are analyzed, and the number of generated preconditions/exceptional post-conditions (columns 4/5). Note that the search space of IR tokens used in C2S is collected only from project JDK. However, specifications for other projects can be generated using C2S, meaning that C2S is cross-project.

4 EVALUATION

We have implemented a prototype C2S and empirically evaluated C2S to address the following questions:

**RQ1:** How effective is C2S in synthesizing formal program specifications from NL comments?

**RQ2:** How does C2S compare with state-of-the-art approaches in improving dynamic testing?

**RQ3:** How useful is C2S in improving static taint analysis?

The evaluation was conducted on a machine with Intel(R) Core(TM) i5-8259U CPU (2.30GHz) and 8GB main memory. The operating system is macOS High Sierra 10.13.6, and the JDK version is 8.

4.1 Data Collection

To prepare the search space of tokens for synthesizing specifications, we collected all available JML method specifications and their corresponding NL comments. All the specifications are composed for the project JDK [6]. In total, we have 3,547 couplings between comments and specifications. On average, each NL word is mapped to 18 IR tokens. The time used to synthesize specifications for each comment is on average 26.40s.

4.2 Effectiveness in Specification Generation

To answer RQ1, we synthesize specifications for 5 frequently-used Java libraries with well-maintained documentation, namely JDK [6], Apache Commons Collections [1], Guava [4], GraphStream [3], and JGraphT [7]. The results are shown in Table 2, which presents the project (column 1), the number of classes/methods (columns 2/3) that are analyzed, and the number of generated preconditions/exceptional behavior post-conditions (columns 4/5). Note that the search space of IR tokens used in C2S is collected only from project JDK. However, specifications for other projects can be generated using C2S, meaning that C2S is cross-project.

From Table 2, we make a few observations. Firstly, the number of normal post-conditions is a bit more than the number of methods. This is because executing one method can lead to multiple effects. Take method `add(int index, Object element)` of class `ArrayList` as an example. If it executes normally, it would achieve at least the following two effects: 1) The receiver list contains the parameter `element`; and 2) The element at the position specified by the parameter `index` equals to the parameter `element`. C2S generates a specification for each effect. Secondly, the number of preconditions is less than that of exceptional post-conditions except for project Apache Commons Collections. Based on our analysis on these comments, we observe that, in many projects, developers rarely comment on preconditions although normally the negation of the exception triggering condition could be considered as essentially a precondition.

To further answer RQ1, we measure precision and recall of C2S in synthesizing specifications. Like other projects of deriving specifications from comments [15, 20, 43], there is no ground truth for

---

**Table 2: Specifications Generated**

<table>
<thead>
<tr>
<th>Project</th>
<th>#Class</th>
<th>#Method</th>
<th>#Pre</th>
<th>#Except Post</th>
<th>#Nor Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK 8.0</td>
<td>10</td>
<td>201</td>
<td>44</td>
<td>99</td>
<td>368</td>
</tr>
<tr>
<td>Commons Collections 4.1</td>
<td>27</td>
<td>170</td>
<td>140</td>
<td>115</td>
<td>187</td>
</tr>
<tr>
<td>Guava 19</td>
<td>8</td>
<td>81</td>
<td>10</td>
<td>13</td>
<td>98</td>
</tr>
<tr>
<td>GraphStream 1.3</td>
<td>4</td>
<td>25</td>
<td>0</td>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>JGraphT 0.9.2</td>
<td>15</td>
<td>34</td>
<td>4</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>64</strong></td>
<td><strong>511</strong></td>
<td><strong>218</strong></td>
<td><strong>243</strong></td>
<td><strong>684</strong></td>
</tr>
</tbody>
</table>
ideal specifications. We follow a similar evaluation method to existing work [15], which manually checks generated specifications. We asked 8 developers (6 graduate students and 2 developers from industry) to participate in the manual checking. All the developers had at least four years of programming experience and were acquainted with program comments. In the process, all specifications are checked against source code with the help of corresponding comments. There are two main reasons: 1) specifications are expected to specify code behaviors; and 2) one NL sentence may express different meanings in different contexts (e.g., "returns the first component" in Fig. 1(g) and in Fig. 1(k) introduced in Section 2). As we manually check specifications, it is inevitable to introduce subjectivity. To minimize such subjectivity, we utilized cross verification by assigning each specification to two different developers. When a disagreement occurs, all the developers would involve in an open discussion to resolve it. Moreover, we mix our synthesized specifications and specifications generated using the other three approaches, and thus the developers are unaware of whether a specification is generated using our approach or not. Note that @tComment aims at detecting comment-code inconsistencies, meaning that it does not assume the correctness of source code. In order to have a fair comparison, during the study, the users were instructed to preclude cases in which specifications are inconsistent with code due to inconsistencies between comments and code.

If a specification is inconsistent with the source code, it is considered as false positive. If we fail to generate a specification for a comment, we consider there is a false negative. Note that there are comments that do not have corresponding specifications (e.g., those explaining time, authors, and implementation details). We preclude such comments. Specifically, a specification is correct (C) when it is consistent with the corresponding source code; it is wrong (W) when it is inconsistent. For example, the specification

\[\forall x : \text{int} \in \text{domain}(x), e_1 \rightarrow e_2\]  

is a correct specification after Algorithm 2.

Table 3: Specification Synthesis Precision and Recall

<table>
<thead>
<tr>
<th>Tool</th>
<th>Pre</th>
<th>Except Post</th>
<th>Normal Post</th>
<th>Overall</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
<td>F</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@tComment</td>
<td>0.98</td>
<td>0.64</td>
<td>0.60</td>
<td>0.18</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>0.93</td>
<td>0.26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toradocu</td>
<td>n.a.</td>
<td>0.00</td>
<td>0.58</td>
<td>0.42</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>0.58</td>
<td>0.41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jdoctor</td>
<td>0.94</td>
<td>0.91</td>
<td>0.91</td>
<td>0.77</td>
<td>0.66</td>
<td>0.39</td>
<td>n.a.</td>
<td>0.85</td>
<td>0.76</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2S</td>
<td>0.98</td>
<td>0.97</td>
<td>0.98</td>
<td>0.91</td>
<td>0.90</td>
<td>0.92</td>
<td>0.88</td>
<td>0.96</td>
<td>0.91</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the return-related normal post-conditions, Jdoctor’s precision and recall are much lower than those of C2S since return comments are too complicated for pattern-matching and lexical similarity to work well.
C2S is the only technique capable of generating non-return-related normal post-conditions. For example, C2S can generate the desired specification `this.get(index) == object` for the comment “Sets the value at the specified index avoiding duplicates.”.

Finally, we want to point out that there are still comments beyond the capabilities of C2S. For example, C2S cannot handle the comment “Returns the n'th item down (zero-relative) from the top of this stack without removing it.”, limited by the incompleteness of existing specifications.

The overall precision of 0.96 and the recall of 0.91 illustrate that C2S is more effective in translating NL comments to formal specifications than the state-of-the-art approaches.

Our generated specifications are publicly available [2].

### 4.3 Improving automatic test case generation

To answer RQ2, we conduct an experiment to show how our specifications improve automatic test cases generation by Randoop compared with Jdoctor’s specifications. We choose Jdoctor because of its diverse specification types. For example, @tComment and Toradocu cannot generate post-conditions for normal behaviors while Jdoctor can. To make a comprehensive comparison, we choose to compare C2S with Jdoctor. This experiment follows Jdoctor’s experiment setup.

Randoop is an automatic testing tool that explores method sequences randomly for a given class and checks whether executing these sequences would violate default specifications as well as user-provided specifications that describe expected behaviors. Randoop outputs two types of test cases, namely failing ones that reveal potential bugs, and passing ones that are used as regression tests.

Randoop has limitations in supporting our generated specifications. To address this, we enhance Randoop in three aspects, and we refer to our enhanced Randoop as C2SRandoop. Firstly, C2SRandoop allows adding oracles between program statements whereas Randoop only adds oracles after the last statement, making it impossible to check some properties before that. For instance, the property `list.isEmpty()` holds after executing the statement `list.clear()`, and we need to add this oracle after the invocation. Otherwise, `list` may be updated by following statements like `list.add("paper")` and it would become invalid to test the property `isEmpty()` after the last statement. Secondly, we modify Randoop to generate oracles for void methods whose specifications are skipped. For example, Randoop does not generate assertions to test the void-return method `add(int index, E element)` of class `ArrayList`, and thus its properties like `this.contains(element)` cannot be checked. Thirdly, Randoop does not accept some of our specification types including implication expressions, forall expressions and expressions containing keyword `yold`. Based on the rules shown in Fig. 5 (introduced in Section 3.6), C2SRandoop can generate oracles from these specifications to test target methods.

We use C2SRandoop to generate test cases based on Jdoctor’s specifications and our specifications, and the time limit is set as 15 minutes. Table 4 summarizes the comparison results. The first column lists the projects. The columns `#FC`, `#TA`, `#FA` and `%` respectively show the number of failing cases, the number of true alarms, the number of false alarms and the ratio between false alarms and the number of failing cases. As we manually check the failing test cases, subjectivity might be introduced. To reduce such subjectivity, we mix failing test cases from Jdoctor and C2S, and developers are unaware of whether test cases belong to Jdoctor or C2S. Moreover, false alarms are checked by multiple developers independently. When a disagreement occurs, all 8 developers involve to discuss to resolve it. From the results, we can see that Jdoctor’s specifications lead to a much higher false alarm than our specifications do (46.91% vs. 12.23% on average). This results from two main reasons. The first one is that Jdoctor’s pattern-matching and lexical similarity to identify subjects (e.g., parameters) in comments are relatively inaccurate. The second one is that Jdoctor does not have an automatic approach to eliminate wrong specifications. By contrast, we leverage our specification checker to automatically prune out incorrect specifications. The columns `#NO` shows the number of new oracles generated based on specifications. Multiple test cases may be generated for a method using the same oracle. Such cases are only counted as one oracle. Note that the number of new oracles generated using our specifications is much higher than that of Jdoctor because C2S can generate much more normal post-conditions than Jdoctor.

### 4.4 Improving identifying leak paths

To answer RQ3, we leverage FlowDroid [12] to conduct taint analysis on 10 Android applications to detect undesirable information leak paths. FlowDroid does not analyze library functions. Instead, it accepts `taint wrappers` to model the information flow of library methods to achieve more precise results. Consider the method `remove(int)` of class `ArrayList`, we can specify the flow from the first parameter of `remove(int)` to the return value and thus FlowDroid can use the flow to analyze code that invokes `remove(int)`, without analyzing the body of the method. Our specifications describe relations between parameters and the return value and thus can be utilized to extract information flow model of a library method.

We limit the sinks to Internet access and files writing, and set a 30 minutes timeout for each app in our experiments. We ran FlowDroid on 10 Android apps in three modes: 1) without using any taint wrapper; 2) with taint wrappers containing data-flow information extracted from Jdoctor’s specifications; and 3) with taint wrappers from our specifications. We compare the number of reported information leak warnings and the performance, which are summarized in Table 5. The first column lists the apps, among which, some are from the DroidBench micro-benchmark suite such as `ArrayAccess1` and they are selected since they use library methods. In order to demonstrate our specifications are beneficial to identify leak paths for different kinds of applications, including open-source and commercial software, benign applications and malwares, we classify applications that use these library methods

<table>
<thead>
<tr>
<th>Project</th>
<th>Jdoctor</th>
<th>C2S</th>
<th>#FC</th>
<th>#TA</th>
<th>#FA</th>
<th>%</th>
<th>#NO</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK 8.0</td>
<td>40</td>
<td>48</td>
<td>9</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Collections 4.1</td>
<td>30</td>
<td>40</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Guava 19</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GraphStream 1.3</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>KGraphT 0.9.2</td>
<td>114</td>
<td>114</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>356</td>
<td>356</td>
<td>20</td>
<td>135</td>
<td>13</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 4: Test Case Generation Improvement

Table 5: Comparing with Jdoctor
into the aforementioned categories, and randomly pick some out of each category.

The number of leak paths and the time (in seconds) used to do the analysis in the three experiments are shown in columns #P and #T. The #ATP columns 6 and 9 present the additional number of true leak paths that are identified using Jdoctor’s specifications and our specifications but cannot be identified when no taint wrapper is used. In summary, the results illustrate that with our specifications, 50 more leak paths can be identified for 8 apps within an acceptable time that is comparable to the analysis time when no taint wrapper is used, while only 7 more leak paths can be identified for 2 apps using Jdoctor’s specification.

5 DISCUSSION

5.1 Threats to Validity

One threat to external validity is that it is possible the space of tokens that compose specifications collected from JDK would cause low accuracy when synthesizing specifications for other projects. To mitigate this, we introduce an intermediate representation to capture the expected behaviors of a method which can be instantiated into a concrete specification for a target method. Another threat lies in that our subject projects and classes might not be representative of true practice. To minimize this, we conducted the evaluation on 5 representative projects that provide diverse functionalities from graph handling to efficient data structures.

The threat to internal validity is the accuracy of our C2S in synthesizing specifications. To alleviate this, we verified the correctness of each generated specification by running developer-written test cases. In addition, part of our evaluation, each specification was manually checked against source code by two developers.

5.2 Generality

Our technique assumes good quality comments. Empirical studies on comments/documentation like [26, 31] have demonstrated over 50% comments/documentation are of good quality and useful in practice. Furthermore, such assumption/limitation is general for most existing work that extracts information from comments like [32, 40, 42, 47, 53]. Despite such limitation, these existing efforts and C2S, have shown that comments in existing projects can be used to improve various aspects of software engineering.

The value of C2S lies in generating specifications for methods whose specifications do not exist or are incomplete. It achieves the goal by learning from a small number of existing specifications (for alignments). Our technique is general in principle and it is a valuable step towards automated specification generation, which is difficult in general. Although we did not present in the paper, C2S works well for C# documentation, and we believe it can be applied to derive specifications for other programming languages.

6 RELATED WORK

There are many efforts of generating specifications from source code or natural language comments, based on static analysis techniques [16, 17, 36, 48–50], dynamic analysis approaches [13, 18, 19, 30], mining large-scale repositories [29, 35, 39], and NLP techniques [15, 20, 32, 43, 54].

Our work is closely related to approaches that infer specifications by analyzing comments written in a natural language. These approaches [15, 20, 32, 41–43, 54] extract specifications from comments by matching handcrafted patterns. Specifically, [53] builds an automation based on the predefined specification template to infer a resource specification to detect bugs. ALICS [32] generates procedure pre/post-conditions that are relevant with strings, integers, null, return and exceptions. iComment [41] extracts usage rules, i.e., lock-related and call-related rules, to detect bugs or bad comments. Similarly, [54] generates parameter constraints to detect directive defects. aComment [42] generates interrupt-related annotations to detect concurrency bugs. @tComment [43] infers null-value related properties of method parameters to detect comment-code inconsistencies. ToradoC [20] synthesizes conditions that can trigger exceptions to create test oracles for exceptional behaviors, and Jdoctor [15] translates code comments to procedure specifications to generate better test cases. Unlike these textual pattern matching approaches, our work does not require predefined patterns that demand manual efforts and may be incomplete. There are other approaches that combine NLP and ML techniques to analyze comments. In [33], researchers use statistical machine translation techniques to translate exception-related documentation to code. Our work leverages alignments which is part of statistical machine translation techniques, but we focus on formal specifications and we generate specifications of various perspectives in addition to exceptions.

7 CONCLUSION

We propose an automatic technique to derive formal program specifications from method NL comments by assembling primitive tokens guided by specification syntax and properties of the target method. We develop a prototype C2S. Our experiments show that C2S can derive specifications efficiently and effectively, with 0.96 precision and 0.91 recall, substantially outperforming the state-of-the-art like Jdoctor. We leverage the generated specifications in a number of software engineering tasks including static taint analysis. The results show our specifications can improve these tasks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part by NSF-China 61802166, 61972193 and 61802171, DARPA FA8650-15-C-7562, NSF 1748764, 1901242 and 1910300, ONR N000141410468 and N000141712947, and Sandia National Lab under award 1701331. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.
REFERENCES


