# Learning-Based Extraction of First-Order Logic Representations of API Directives

Mingwei Liu*
Fudan University
China

Xin Peng*†
Fudan University
China

Andrian Marcus
The University of Texas at Dallas
USA

Christoph Treude
The University of Adelaide
Australia

Xuefang Bai*
Fudan University
China

Gang Lyu*
Fudan University
China

Jiazhan Xie*
Fudan University
China

Xiaoxin Zhang*
Fudan University
China

## ABSTRACT

Developers often rely on API documentation to learn API directives, *i.e.,* constraints and guidelines related to API usage. Failing to follow API directives may cause defects or improper implementations. Since there are no industry-wide standards on how to document API directives, they take many forms and are often hard to understand by developers or challenging to parse with tools.

In this paper, we propose a learning based approach for extracting first-order logic representations of API directives (*FOL directives* for short). The approach, called LeadFOL, uses a joint learning method to extract atomic formulas by identifying the predicates and arguments involved in directive sentences, and recognizes the logical relations between atomic formulas, by parsing the sentence structures. It then parses the arguments and uses a learning based method to link API references to their corresponding API elements. Finally, it groups the formulas of the same class or method together and transforms them into conjunctive normal form. Our evaluation shows that LeadFOL can accurately extract more FOL directives than a state-of-the-art approach and that the extracted FOL directives are useful in supporting code reviews.

## CCS CONCEPTS

• **Software and its engineering → Documentation**; **Maintaining software**; **Software libraries and repositories**.

*M. Liu, X. Peng, X. Bai, G. Lyu, J. Xie and X. Zhang are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, and the Shanghai Institute of Intelligent Electronics & Systems, China.
†X. Peng is the corresponding author (pengxin@fudan.edu.cn).


*ESEC/FSE '21, August 23–28, 2021, Athens, Greece*

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468618

## KEYWORDS

Directive, First Order Logic, API Documentation

**ACM Reference Format:**
Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Xuefang Bai, Gang Lyu, Jiazhan Xie, and Xiaoxin Zhang. 2021. Learning-Based Extraction of First-Order Logic Representations of API Directives. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468618

## 1 INTRODUCTION

Many software development tasks need to be accomplished by reusing Application Programming Interfaces (APIs) provided by libraries or frameworks. During this process, developers often rely on API documentation to identify relevant APIs and learn API usages [5, 7, 21]. API documentation provides a variety of different types of information such as, functionality descriptions, directives, purpose and concept explanations [9]. Among them, *directives* are especially important for the correct usage of APIs. *API directives* make developers aware of constraints and guidelines related to the usage of an API [13]. Failing to follow API directives may lead to defects or improper implementations.

Although well-maintained API documentation exists, it is often not easy for developers to effectively access relevant knowledge on how to use an API [5, 7, 8]. This problem is evidenced by the fact that API misuses are common and developers often ask API related questions in online forums such as, Stack Overflow. To improve the accessibility of API directives, existing research [5, 7, 8] extracts directive sentences from API documentation. The extracted API directives can be recommended to developers in an on-demand way or a push way for the APIs used in code.

To further support automated tasks such as, the detection of API misuses [24] and documentation defects [32], the API directives need to be transformed into a machine-processable form. For this purpose, researchers have proposed approaches for extracting different kinds of formally expressed API directives from documentation, *e.g.,* resource specifications [31], temporal and constraints [17], parameter constraints [25] and call-order/condition-checking [20]. Recently, Zhou *et al.* [32] proposed an approach called DRONE

that extracts API directives represented in first-order logic (FOL) formulas. These approaches rely on recurrent linguistic patterns to extract formal expressions of API directives and, thus, are able to extract only certain types of directive types, with limited accuracy.

API directives in natural language can be expressed in many different forms, with complex logical structure. Hence it is challenging to recognize API directives and parse them into formal representations such as FOL formulas. First, an API directive may involve complex logical structures consisting of multiple clauses. For example, the return value directive for `javax.swing.UIDefaults.getIcon(Object)` is "if the value for `key` is an `Icon`, return the `Icon` object; otherwise return `null`". For understanding this directive, it is necessary to recognize the clauses and their logical relations implied by the connectors (*e.g.,* *if*, *otherwise*, *and*, *or*). Second, the predicate of a clause can be expressed in many different ways. For example, instead of stating that a parameter is non-negative, a directive may state that passing a negative parameter is not allowed; instead of explicitly requesting that a value is an instance of `javax.swing.Icon`, a directive may simply state that the value is an Icon. Third, the arguments of a clause may involve API elements that are mentioned with different kinds of pronouns or aliases. For example, a parameter named "`obj`" may be mentioned as "the second parameter" or "the object"; and the return value of `java.lang.StringBuffer.length()` can be mentioned as "the length of the sequence". For understanding such API directives, it is necessary to employ complex linguistic anaphora resolution techniques.

In this paper, we aim to automatically identify API directives and transform them into FOL formulas in conjunctive normal form, which consist of atomic formulas and logical operators (*i.e.,* conjunction, disjunction, implication). The types of atomic formulas and their forms in natural language expressions of API directives are essential for the extraction of FOL representations of API directives. To investigate the types and forms of atomic formulas, we studied 729 directive sentences sampled from the reference documentation of JDK 1.8. The study resulted in 24 predicates that can be used in the atomic formulas of API directives, together with several types of arguments involved in the formulas.

Based on these findings, we propose a learning based approach for extracting first-order logic representations of API directives (*FOL directives* for short) from documentation, which we call LeadFOL (**L**earning based **E**xtraction of **A**PI **D**irectives in **FOL**). Given API documentation (*e.g.,* reference documentation or tutorial), LeadFOL extracts description sentences and trains a sentence classifier to identify directive sentences. To extract atomic formulas from clauses, it uses a joint learning method, which trains a sequence tagging model to identify the predicates and arguments involved in the clauses. It recognizes the logical relations between atomic formulas by parsing the structures of the corresponding directive sentences. Then, it also recognizes the constants, expressions, and API elements involved in the arguments. Specifically, it uses a learning based method to link API references to the corresponding API elements. Finally, it groups the formulas of the same class or method together and transforms them into conjunctive normal form.

We empirically evaluated the effectiveness and usefulness of LeadFOL. First, we intrinsically evaluated the key steps of LeadFOL independently, *i.e.,* directive sentence identification, atomic formula extraction, logical relation recognition, and argument parsing. Then, we evaluated the accuracy of the extracted FOL directives by comparing them with the results of DRONE [32]. We also evaluate the applicability of LeadFOL on APIs from other libraries. Finally, we evaluate the usefulness of the FOL directives extracted by LeadFOL, in supporting developers during code reviews. The results show that LeadFOL accurately extracts more FOL directives than DRONE [32] and the extracted FOL directives are useful in supporting code reviews.

## 2 PREDICATE IDENTIFICATION

We investigate different types atomic formulas involved in API directives and how they are expressed in API documentation. To this end, we sampled a set of directive sentences from the reference documentation of JDK 1.8[1] and conducted a qualitative analysis, using open coding.

### 2.1 Data Preparation

As in previous studies [13, 32], we obtained the API documentation from the source code of the target API library by extracting the comments before class, interface, and method declarations. Then, we extracted the following five types of text for further analysis:

- *Class Description*: leading text of class comments that appears before annotations (*e.g.,* "@author").
- *Method Description*: leading text of method comments that appears before annotations (*e.g.,* "@param").
- *Exception Description*: text in exception annotations (*i.e.,* "@throws", "@exception").
- *Parameter Description*: text in parameter annotations (*i.e.,* "@param").
- *Return Value Description*: text in return value annotations (*i.e.,* "@returns").

We cleaned the extracted comments by removing delimiters (*e.g.,* "/**", "*/"), HTML tags (*e.g.,* "<tt></tt>"), and annotations (*e.g.,* "{@link}"). Then, we split the text into sentences using the Spacy NLP library.[2]

For identifying candidate directive sentences we used a set of directive related keywords. Monperrus *et al.* [13] report a set of keywords and regular expressions that are likely to reveal directives. Zhou *et al.* [32] define 64 linguistic patterns for parameter constraints (*e.g.,* "`[Parameter]` can not be null" ), based on which we extracted a set of directive related keywords. We merged the two sets and obtain 86 keywords and regular expressions (*e.g.,* "must", "only", "note") for identifying candidate directive sentences. The complete set can be found in our replication package [2].

Then, we sampled a set of candidate sentences that include one of the keywords or conform to one of the regular expressions based on two criteria: (1) sample at least 200 sentences from each of the five types of text; and (2) sample at least 5 sentences for each keyword or regular expression. The sampling resulted in 1,167 sentences, after removing duplicate ones. Two of the authors independently examined the sampled sentences to annotate whether they express API directives or not. They achieved a Cohen's Kappa agreement [10]

[1] https://www.oracle.com/java/technologies/javase/javase8u211-later-archive-downloads.html
[2] https://spacy.io/

of 0.807, *i.e.,* almost perfect agreement. For the sentences that were annotated differently, a third author served as arbiter and made the final decision. Finally, we obtained 729 directive sentences, which we analyzed further.

## 2.2 Coding Protocol

The coding was done in two phases: seed coding phase and sentence annotation phase. The purpose of the seed coding phase is defining a set of initial predicates to facilitate the subsequent sentence annotation. The purpose of the sentence annotation phase is identifying the implicit atomic formulas from the sampled directive sentences.

During the seed coding phase, we identified the following 13 predicates from the 64 linguistic patterns reported in [32]: *Equal*, *NotEqual*, *MayBe*, *Greater*, *Less*, *GreaterOrEqual*, *LessOrEqual*, *Negative*, *Positive*, *NonNegative*, *NonPositive*, *InstanceOf*, *NotInstanceOf*. The descriptions of these predicates can be found in Table 1.

In the sentence annotation phase, two of the authors independently annotated the atomic formulas in the 729 directive sentences. For each sentence the annotators understood its meaning, based on its context in the documentation, and identified the implicit formulas based on the currently defined predicates. They split some sentences into one or multiple clauses that can be transformed into atomic formulas and identified the predicate implied and the arguments involved in each clause.

If the identified formulas are different, a third author joined and started a discussion to reach an agreement. To avoid different interpretations of the same clause, we defined the following two guidelines for the annotation:

(1) Infer the missing arguments based on the context, *e.g.,* in the if clause "If `null`, this method throws an exception";
(2) If possible use a single predicate rather than multiple ones, *e.g.,* use *GreaterOrEqual* instead of *Greater* and *Equal*.

If the identified formula involves an undefined predicate, the three authors discussed whether to accept it as a new predicate, based on the following two criteria:

(1) Only define predicates that are commonly used by different APIs. For example, we will not define a predicate *Sorted* for the clause "The indices must be in sorted order, from lowest to highest", as "sorted" is not common.
(2) Only define predicates that have clear evaluation criteria. For example, we will not define a predicate *NotValid* for the clause "if the index is not valid", as it is vague.

Whenever a new predicate was identified, the annotators checked whether the annotations that have been made needed updating to align with the new predicate definitions. Note that not all 729 directive sentences were captured using atomic formulas.

## 2.3 Results

The sentence annotation phase resulted in 11 new predicates. These predicates together with the 13 predicates identified in the seed coding phase constitute the predicates that are used to express the atomic formulas of API directives. Table 1 shows the 24 predicates together with their definitions and examples. The text in italics in the examples indicates the arguments involved in the predicates.

These predicates can be classified into four categories: *value* (16), *type* (2), *action* (2), and *usage* (4). *Value* predicates specify the range of a given value (*e.g.,* parameter, return value). *Type* predicates specify whether an object is the instance of a class/interface. *Action* predicates specify the actions (*e.g.,* throw exception) that a method may take in a certain situation. *Usage* predicates specify how a class or method should be used, including the methods that need to be overridden, deprecated classes, replaced methods, and equivalent methods. Compared with the formulas defined in prior related research [32], the formulas identified in our study include additional *action* and *usage* predicates and more *value* predicates (*i.e., MayNotBe*, *MayNegative*, *MayPositive*, *MayNonnegative*, *MayNonpositive*). The *MayXXX* predicates indicate something that may occur and should be considered in API usage. *e.g.,* "Nullable" can be regarded as a specialization of *MayBe*.

From the annotated atomic formulas, we identified the following types of arguments:

- *Literal Constant*: constants of different types (*e.g.,* number, string, boolean), including the special value "`null`";
- *API Elements*: API classes/interfaces (including exception and error classes), their properties and methods, and the parameters and return values of their methods;
- *Mathematical Expressions*: mathematical expressions composed by the above elements, operators (*e.g.,* "+", "-"), and mathematical functions (*e.g., log*).

Note that some predicates are opposite or can be expressed by other predicates. For example, *NotEqual* is the negation of *Equal*; and *Negative* can be expressed using *Less*. We reserve these equivalent predicates to facilitate the recognition of atomic formulas from directive sentences, as their expressions in natural language are quite different.

# 3 APPROACH

Given an API library and its documentation, LeadFOL extracts FOL directives for API classes/interfaces and methods as follows:

- **Directive Sentence Identification**: Extract description sentences from API documentation and identify directive sentences from them.
- **Atomic Formula Extraction**: Extract atomic formulas from each directive sentence by recognizing the predicates and arguments.
- **Logical Relation Recognition**: Recognize the logical relations between the extracted atomic formulas by parsing the structure of the corresponding directive sentence.
- **Argument Parsing**: Parse the arguments in the extracted atomic formulas to recognize the involved constants, expressions, and API elements.
- **Formula Normalization**: Group the formulas of the same API class/interface or method together and transform them into conjunctive normal form.

The following subsections detail these steps. We use the JDK method `StringBuffer.insert(int, char)`[3] as a **running example** throughout the section to illustrate the approach.

[3]https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html

**Table 1: Predicates for FOL Expressions of API Directives**

| Category | Predicate | Definition | Example | Predicate | Definition | Example |
|---|---|---|---|---|---|---|
| Value | Equal($v_1$, $v_2$) | $v_1$ == $v_2$ | If *the obj* is *null*... | NotEqual($v_1$, $v_2$) | $v_1 \neq v_2$ | *The param* will never be *null*. |
| | MayBe($o$, $v$) | $o$ *may be* $v$ | *The applet* may be *null*. | MayNotBe($o$, $v$) | $o$ *may not be* $v$ | *It* may not be *null*. |
| | Greater($v_1$, $v_2$) | $v_1 > v_2$ | *The length* is greater than *0*. | Less($v_1$, $v_2$) | $v_1 < v_2$ | If *size* is less than *0*... |
| | GreaterOrEqual($v_1$, $v_2$) | $v_1 \geq v_2$ | *the starting offset >= 0* | LessOrEqual($v_1$, $v_2$) | $v_1 \leq v_2$ | If *init* is illegal (*<=0*)... |
| | Negative($v$) | $v < 0$ | If *this value* is negative... | Positive($v$) | $v > 0$ | *The number* must be positive. |
| | MayNegative($v$) | $v$ may be negative | Calling seek with a negative *index* is legal. | MayPositive($v$) | $v$ may be positive | *The value* may be positive. |
| | Nonnegative($v$) | $v >= 0$ | Passing negative *parameter* is not recommended. | Nonpositive($v$) | $v <= 0$ | If $m$ is not positive... |
| | MayNonnegative($v$) | $v$ may be nonnegative | *The returned number* may be a non-negative decimal value. | MayNonpositive($v$) | $v$ may be nonpositive | A non-positive *parameter* is allowed. |
| Type | InstanceOf($i$, $t$) | $i$ is an instance of $t$ | If *this node* is of type *Document*... | NotInstanceOf($i$, $t$) | $i$ is not an instance of $t$ | If *the permission* is not a *BasicPermission*... |
| Action | Throw($m$, $e$) | $m$ throws an exception $e$ | *This method* will throw *a NullPointerException*... | Return($m$, $v$) | $m$ returns $v$ | *This method* always returns *true*. |
| Usage | NeedOverride($m$) | $m$ needs to be overridden in subclasses | *This method* needs to be overridden by extending classes. | Deprecated($c$) | $c$ is deprecated | *This class* is deprecated. |
| | Replace($n$, $o$) | $n$ replaces $o$ | *This* is a replacement for *sun.awt.AppContext*. | Equivalent($m_1$, $m_2$) | $m_1$ and $m_2$ are equivalent | *This method* is equivalent to *ofEpochDay(long)*. |

## 3.1 Directive Sentence Extraction

Similar to the data preparation in the predicate identification study (see Section 2.1), this step extracts five types of text from the API documentation (*i.e.,* class descriptions, method descriptions, exception descriptions, parameter descriptions, and return value descriptions), cleans the text and splits the text into sentences. Then, we complete the obtained sentences and identify directive sentences from them.

The purpose of sentence completion is to facilitate subsequent analysis. To this end, we use Spacy to analyze the sentences by tokenization, POS (part-of-speech) tagging, and dependency parsing, and complete the sentences accordingly.

- If a sentence has no subject, then we add a subject according to the sentence type: "*This class*" for class description; "*This exception*" for exception description; "*This parameter*" for parameter description; and "*This method*" for method description and return value description.
- If a sentence has no predicate, then we add a predicate according to the sentence type: "*throws*" for exception description, "*returns*" for return value description, and "*is*" for the others.
- If a sentence is an exception description and only has a conditional clause, then we add "*This method throws an exception*" as the main clause.

Similar to Liu *et al.* [8], we train a sentence classifier to identify directive sentences. We use FastText[4] to train a binary classifier using the following training data obtained from three sources: 1,167 sentences annotated in our predicate identification study (see Section 2.1); 8,347 sentences annotated by Liu *et al.* [8]; 6,778 sentences randomly selected from the JDK 1.8 reference documentation and annotated by four of the authors. As a result, the training set includes 8,314 directive sentences and 7,978 non-directive sentences.

**Running example.** For `StringBuffer.insert(int, char)`, Lead-FOL extracts the following two directive sentences:

**DS1:** *The offset argument must be greater than or equal to 0, and less than or equal to the length of this sequence.*

**DS2:** *This method throws an exception if the offset is invalid.*

DS1 is extracted from the method description; DS2 is extracted from the exception description and obtained after completing the original sentence "if the offset is invalid."

[4]https://github.com/facebookresearch/fastText

## 3.2 Atomic Formula Extraction

As discussed in Section 1, the predicates and arguments are expressed in many different forms in directive sentences, *e.g.,* "the index must be greater than or equal to 0", "index >=0", or "index < 0 is illegal". It is thus hard to recognize the predicates and arguments using linguistic patterns. Therefore, we design a joint learning method for extracting atomic formulas by recognizing the predicates and their arguments in a directive sentence. An intuitive method for this problem is to treat predicate recognition and argument recognition as two subtasks and design an independent model for each of them. However, this method would neglect the relevance between the two subtasks since the results of one subtask may affect the performance of the other one [30]. In contrast, joint learning methods use a single model to handle multiple relevant subtasks to effectively integrate the information of these subtasks. They have been shown to achieve better results in NLP tasks such as joint extraction of entities and relations [30].

We train a sequence tagging model to implement joint learning for predicate and argument recognition. Sequence tagging [18] predicts the corresponding tag sequence of an observation sequence and is widely used for NLP tasks such as POS tagging, chunking, and named entity recognition (NER). The idea is that each argument of a predicate can be treated as a special entity and the predicate can be regarded as a relation of related entities, thus the recognition of the predicate can be done by recognizing all the involved entities. For example, the predicate *Throw* can be identified if a method entity and a thrown exception entity are recognized in a sentence.

*3.2.1 Tagging Schema.* To use sequence tagging for our purpose, we need to design a tagging scheme that can represent the involved arguments of the predicates defined in Table 1. Our tagging scheme is based on the widely used IOBES scheme [18]: "B" ("Beginning"), "I" ("Inside"), and "E" ("End") respectively indicate the beginning, middle, and end of an entity; "S" ("Single") indicates that the current word itself constitutes an entity; and "O" ("Outside") indicates a normal word. For each argument of a predicate, we define a named entity type and four tags (*i.e.,* "B", "I", "E", and "S"). For example, for the predicate *Throw* we define two entity types *Throw-1* and *Throw-2* for the method and thrown exception respectively and four tags for each entity type (*e.g.,* "Throw-1-B", "Throw-1-I", "Throw-1-E", "Throw-1-S"). To support predicates expressed in passive voice we

include an additional passive form for *Throw*, *Return*, and *Replace* in sequence tagging: *ThrownBy(e, m)*, *ReturnedBy(v, m)*, and *ReplacedBy(o, n)*. As a result, we define 46 types of named entities for the 24 predicates and 184 tags for the entity types together with a special tag "O" ("Outside"). The definitions of these entity types and tags can be found in our replication package [2].

Based on the tagging schema, we provide the following guidelines for sentence annotation:

- each word has exactly one tag;
- the ranges (*i.e.,* between the beginning and the end) of two argument do not overlap;
- the tags of different arguments of the same predicate occur in the same order as they occur in the definition of the predicate in Table 1; and
- an argument should be annotated as complete as possible (*e.g.,* "The offset argument" instead of "offset").

It is possible that multiple clauses share the same argument. For example, the *GreaterOrEqual* clause and the *LessOrEqual* clause of DS1 share the argument "The offset argument". In this case we annotate the shared argument according to the predicate that appears first in the sentence. For example, we annotate the argument "the offset argument" in DS1 according to the *GreaterOrEqual* predicate. We will resolve the shared argument for the other involved predicates later, when we generate the atomic formulas.

**Running example.** The annotations for DS1 and DS2 are given below (GE, LE, TH are the abbreviations of *GreaterOrEqual*, *LessOrEqual*, *Throw*, respectively), which provide a tag for each word. DS1 includes three arguments of two predicates with one shared argument, while DS2 includes two arguments of one predicate.

**DS1:** GE-1-B: *The* GE-1-I: *offset* GE-1-E: *argument* O: *must* O: *be* O: *greater* O: *than* O: *or* O: *equal* O: *to* GE-2-S: *0*, O: *and* O: *less* O: *than* O: *or* O: *equal* O: *to* LE-2-B: *the* LE-2-I *length* LE-2-I: *of* LE-2-I: *this* LE-2-E: *sequence*.

**DS2:** TH-1-B: *This* TH-1-E: *method* O: *throws* TH-2-B: *an* TH-2-E: *exception* O: *if* O: *the* O: *offset* O: *is* O: *invalid*.

*3.2.2 Learning Model.* We use an open source implementation of BERT-BiLSTM-CRF, called Kashgari[5], a deep learning based sequence tagging model, to predict the tags of the words of a sentence. Taking a sentence (a sequence of words) as input, the model predicts for each word a tag from a predefined set. The model includes there layers: BERT layer, Bi-LSTM layer, and CRF layer. The BERT (Bidirectional Encoder Representations from Transformers) layer is an embedding layer that converts each input word to a fixed-size vector using the pretrained language model [4]. BERT is pretrained on a large corpus and its vector representation can capture the semantics of a word in different contexts well. The Bi-LSTM (Bi-directional Long Short-Term Memory) layer further encodes and combines the vector of a word together with its left context and right context to prepare the input for the CRF layer. The CRF (Conditional Random Field) layer produces a tag for each word. It considers the correlations between the tags of neighboring words and jointly decodes the best chain of tags for a given input sentence, thus is suitable for the sequence tagging task.

[5]https://github.com/BrikerMan/Kashgari

*3.2.3 Training.* To prepare training data for the model, we asked four Master students to annotate the sampled directive sentences from the training set of the directive sentence classifier. The annotation was done with the online annotation tool doccano [14]. We conducted a two-hour training session to help the four annotators learn the annotation tool, tagging schema, and annotation guidelines. Each directive sentence was annotated by two annotators, independently. If their annotations were different, one of the authors joined and resolved the conflict using the majority strategy. As a result, we obtained 3,226 annotated directive sentences with 7,045 arguments. For 78.2% of the 3,226 directive sentences, the annotations of two people was identical, *i.e.,* high consensus.

We followed a commonly used data augmentation technique in computer vision [6] and natural language processing [28], which can help train more robust models, particularly when using smaller data sets. Based on the annotated sentences, we used a set of heuristic rules to generate more training data, including:

- replace an argument in a sentence with an argument of the same type (*e.g.,* value, method, class) randomly selected from other sentences;
- exchange two arguments of the same predicate if they have the same type; and
- replace an argument reference with an automatically generated alias (*e.g.,* "`StringBuilder`" is replaced by "this class" or "string builder").

These rules are designed based on our observations of the annotated sentences, by considering the syntactic structures of the sentences and the characteristics of the predicates. The generated data increases the diversity of the training data and at the same time conforms to the human annotations.

Finally, we obtained 11,019 annotated directive sentences with 16,972 annotated arguments as the training data.

*3.2.4 Atomic Formula Generation.* Given a directive sentence, the trained sequence tagging model predicts a tag for each word. Then, we can generate atomic formulas based on the tagging. If all arguments of a predicate are successively found in the tagging, we generate an atomic formula with the predicate and the recognized arguments. Due to the existence of shared arguments, it is possible that a predicate only has its second argument found. In this case, we try to find the closest first argument of other predicates before the current predicate and combine the argument with the second argument of the current predicate to generate an atomic formula.

**Running example.** With the above strategy, we extract the following three atomic formulas for `StringBuffer.insert(int, char)`:
*GreaterOrEqual("the offset argument", "0")*,
*LessOrEqual("the offset argument", "the length of this sequence")*,
*Throw("this method", "an exception")*.

## 3.3 Logical Relation Recognition

This step recognizes the logical relations between the extracted atomic formulas by analyzing relations between the corresponding clauses in the directive sentences. A directive sentence may include multiple clauses connected by conjunctions. In our **running example**, **DS1** includes two clauses connected by "*and*"; **DS2** includes two clauses connected by "*if*". Based on the predicate identification study, we find that a directive sentence can be composed at two

levels. At the higher level, the sentence can have a nested conditional structure using conjunctions like "*if*", "*then*", and "*otherwise*". At the lower level, the clauses of the conditional structure (called composite clauses) can be further composed by simple clauses using conjunctions like "*and*", "*or*", and "*not*". Accordingly, the clause relation analysis can be done in two steps, *i.e.,* conditional structure parsing and clause parsing.

*3.3.1 Conditional Structure Parsing.* Conditional structure parsing is done by linguistic pattern matching. Three of the authors manually analyzed the directive sentences identified in the predicate identification study and iteratively created and merged linguistic patterns for conditional structure parsing. The process resulted in 51 linguistic patterns defined using the following conjunctions: *if*, *then*, *otherwise*, *unless*, *when*, *where*, *since*, *as soon as*, *depend on*, *because*, *while cause*, *indicate*. Table 2 shows a subset of the patterns with examples, where the composite clauses are shown with wavy lines. The complete set of linguistic patterns can be found in our replication package [2]. We define a regular expression for each pattern and parse the conditional structure of a sentence by matching with the regular expressions. During matching, modal verbs before conditional conjunctions will be ignored, for example "will cause" will be treated as "cause".

For each linguistic pattern we define a formula pattern which specifies the logical relations among the composite clauses.

- If a directive sentence matches a linguistic pattern, then the matched clauses are extracted as composite clauses and their logical relations are recorded.
- If the sentence matches no linguistic pattern, then it is treated as a single composite clause for further analysis.
- If the sentence matches more than one linguistic patterns, then we parse it according to the most complete and concrete pattern, *i.e.,* the pattern that has the most conditional conjunctions and the deepest nested structure.

For example, the second example in Table 2 will be parsed according to the second pattern, not the first one.

**Running example.** For `StringBuffer.insert(int, char)`, **DS1** is treated as a single composite clause; **DS2** is parsed into two composite clauses "this method throws an exception" (**DS2-1**) and "the offset is invalid" (**DS2-2**) with an implication relation between them.

*3.3.2 Clause Parsing.* Clause parsing further extracts simple clauses from composite clauses and is done by syntactic structure parsing. Given a composite clause, we use Spacy to analyze its syntactic structure by POS tagging and dependency parsing. If the composite clause has multiple predicates or objects, we split it into multiple simple clauses with only one predicate and one object and complete the subjects of the simple clauses when required. Each simple clause will further be parsed to extract an atomic formula, and the logical relations between simple clauses are determined by the logical conjunctions (*e.g.,* "and", "or") that connect them.

**Running example.** For `StringBuffer.insert(int, char)`, two simple clauses "the offset argument must be greater than or equal to 0" (**DS1-1**) and "the offset argument must be less than or equal to the length of this sequence" (**DS1-2**) are recognized with a conjunction relation between them for **DS1**; **DS2-1** and **DS2-2** themselves are simple clauses for **DS2**.

## 3.4 Argument Parsing

We adopt different argument parsing methods to recognize two different kinds of elements in an argument, *i.e.,* expressions (including literal constants, mathematical expressions) and API elements (including API classes/interfaces, properties, methods, parameters, and return values).

Given an argument, we check whether it is a literal constant or a mathematical expression by matching with predefined regular expressions. If the argument is a mathematical expression, then we iteratively parse its elements.

Given a candidate reference of an API element in an argument, we first check whether it is an API related pronoun like "this class", "this method", and "this parameter". If it is, we link it to the target API element according to the position where the corresponding directive sentence was extracted. Otherwise, we try to recognize the API element based on an automatically constructed API graph. The API graph consists of API elements such as classes/interfaces, properties, methods, parameters, return values and their relationships such as containment, inheritance, implementation, and instantiation. Each API element has a fully qualified name with the first sentence of its description text as its definition. We further generate a list of aliases for it using heuristic rules, such as short name, method name with the parameter list, *etc.*. The complete set of heuristic rules can be found in our replication package [2]. Then we recognize the API element in the following three steps.

**1. Selection of Candidate API Element**. We select candidate API elements for the reference based on both name and context. Name based selection considers the name similarity between the reference and API elements. We normalize the reference by removing stop words and lemmatization and then match it with the names (both the fully qualified names and aliases) of API elements. We do not consider parameter names in the matching, as it is rare that a parameter of a method is referenced outside of the description of the method. If there are no API elements that fully match the reference, we try to match each word of the reference. Context based selection considers the position where the corresponding directive sentence is extracted, that is the class or method.

**2. Candidate Filtering by Type**. We filter out candidate API elements that are not compatible with the required type of the argument. For example, if the reference belongs to the second argument of the predicate *InstanceOf*, it should be a class or an interface. Thus all the candidate API elements that are not classes or interfaces are filtered out.

**3. Learning based Candidate Ranking**. We train a machine learning model to predict the probability of linking for each candidate API element. The top ranked one is chosen as the linked API element. Given an API reference $m$ and a candidate API element $c$, we consider the following five commonly used similarity metrics as the features:

- *Name Lexical Similarity*: the maximum similarity between $m$ and the names of $c$ calculated based on Levenshtein distance [12];

**Table 2: Linguistic Patterns for Conditional Structure Parsing (Partial)**

| Linguistic Pattern | Formula Pattern | Example |
|---|---|---|
| $c_1$ if $c_2$ | $c_2 \rightarrow c_1$ | This method returns true if the list is empty. |
| $c_1$ if $c_2$ otherwise $c_3$ | $(c_2 \rightarrow c_1) \wedge (\neg c_2 \rightarrow c_3)$ | This method returns null if the Component is null; otherwise,returns the passed-in rectangle. |
| if $c_1$ then $c_2$ if $c_3$ otherwise $c_4$ | $((c_1 \wedge c_3) \rightarrow c_2) \wedge ((c_1 \wedge \neg c_3) \rightarrow c_4)$ | If the value is negative infinity, then the output will be "(Infinity)" if the "(" flag is given otherwise the output will be "-Infinity". |
| $c_1$ cause $c_2$ | $c_1 \rightarrow c_2$ | Passing null parameter will cause nullpointer exception to be thown. |

- *Name Cosine Similarity*: the maximum similarity between $m$ and the names of $c$ calculated as the cosine similarity between their vector representations, which are obtained by averaging word vectors produced by Word2Vec;
- *Context Lexical Similarity*: the Jaccard [15] similarity between the definition sentence of $c$ and the context of $m$ (*i.e.,* the surrounding words of $m$ in the directive sentence);
- *Context Cosine Similarity*: the cosine similarity between the vector representations of the definition sentence of $c$ and the context of $m$, which are obtained by averaging word vectors produced by Word2Vec;
- *Graph Similarity*: the graph similarity between $c$ and the API element where $m$ is extracted from, which is calculated as $1/(d+1)$ where $d$ is the shortest distance between them in the API graph.

We use the 100-dimensional Word2Vec [11] model pretrained on the Wikipedia corpus [3] and tune it based on the corpus of the JDK 1.8 documentation using Gensim [1]. Based on the above five features we train a SVM (Support Vector Machine) model using scikit-learn[6]. To prepare the training data we manually annotated 246 API related arguments from the annotated formulas in the predicate identification study. For each positive training sample, we randomly select five other API elements from the link candidates of the target reference as negative training samples. We use Word2vec embeddings instead of BERT embeddings, because BERT embeddings can't be used with cosine similarity or Manhattan/Euclidean distance, as indicated by existing study [19].

**Running example.** For the four atomic formulas extracted for `StringBuffer.insert(int, char)`, "the offset argument" is linked to the *offset* parameter of the method; "0" is recognized as a constant value; "the length of this sequence" is linked to the return value of `java.lang.StringBuffer.length()`; "this method" is linked to the method itself; and "an exception" is linked to `java.lang.StringIndexOutOfBoundsException`.

## 3.5 Formula Normalization

We group the extracted formulas for each class or method and transform them into conjunctive normal form in four steps.

**1. Atomic Formula Normalization**. Transform the formulas expressed in passive voice (*i.e., ThrownBy*, *ReturnedBy*, and *ReplacedBy*) into the corresponding forms of active voice (*i.e., Throw*, *Return*, and *Replace*). Transform the formulas of negative expression into the corresponding forms of positive expression if exists. For example, $\neg$*LessOrEqual*($v_1$, $v_2$) is transformed into *Greater*($v_1$, $v_2$).

**2. Composite Formula Normalization**. For each directive sentence, generate a sentence formula by connecting all the atomic formulas extracted from its simple clauses based on the recognized logical relations. Then, generate an overall formula for the class or method by connecting all its sentence formulas with conjunctions (*i.e., AND*) and transform the overall formula into its conjunctive normal form. Note that it is possible that a simple clause does not include any predefined predicate due to its vagueness (*e.g.,* "the offset is invalid") or failed predicate recognition. In this case, we treat the whole clause as an atomic formula with a general predicate *Statement*, which means that the stated fact holds.

In our **running example**, a *Statement* formula will be generated for the clause **DS2-2** ("the offset is invalid").

**3. Implicit Formula Derivation**. For each clause of the compound formula in conjunctive normal form, generate derived clauses according to the following rules:

- A condition that will lead to exceptions is not allowed: if there is $F \rightarrow$ *Throw(m, e)*, generate a derived clause $\neg F$.
- A value range mentioned in a condition is allowed: if there is *F1* $\rightarrow$ *F2* and *F1* is an *Equal*, *NotEqual*, *Negative*, *Positive*, *Nonnegative*, *Nonpositive* predicate, generate a derived clause with the corresponding "MayBe" predicate. For example, if *F1* is *Equal*($v_1$, $v_2$), the derived formula will be *MayBe*($v_1$, $v_2$).
- A returned value mentioned in a conditional expression is possible: if there is $F \rightarrow$ *Return(m, v)*, generate a derived clause *MayBe(m.RV, v)* where *m.RV* denotes the return value of the method.

**4. Equivalent Formula Merging**. Merge different clauses of the compound formula that express equivalent formulas, including equivalent formulas with different predicates or argument orders, *e.g., Greater*($v_1$, $v_2$) and *Less*($v_2$, $v_1$).

For our **running example** `StringBuffer.insert(int, char)`, we obtain the following FOL directives, where *offset* is the parameter of the method and *length* is the return value of `java.lang.StringBuffer.length()`:

*GreaterOrEqual(offset, 0)* $\wedge$ *LessOrEqual(offset, length)* $\wedge$ *(Statement("the offset is invalid")* $\rightarrow$ *Throw(*`StringBuffer.insert(int, char)`, `java.lang.StringIndexOutOfBoundsException`*))*

# 4 EVALUATION

We conduct a series of experimental studies to evaluate the effectiveness and usefulness of LeadFOL by answering the following four research questions:

**RQ1**: What is the intrinsic quality of the key steps of LeadFOL?

**RQ2**: How does LeadFOL compare with state-of-the-art for FOL directives extraction?

**RQ3**: How does LeadFOL generalize to non-JDK APIs?

**RQ4**: How does LeadFOL support code reviews?

## 4.1 Quality of Key Steps (RQ1)

We evaluate the quality of the results of the main steps of the approach (cf. Section 3), except formula normalization. The latter is

[6]https://scikit-learn.org/

based on deterministic rules whose quality solely depends on the output of previous steps.

*4.1.1 Directive Sentence Identification.* We conducted a ten-fold cross validation with the 16,103 sentences annotated for the training of our sentence classifier. We trained the FastText model with the default configurations of the official implementation. We calculated the following four metrics: precision, recall, and F1-measure of directive sentence recognition, and accuracy of sentence classification (directive or not). The average results of the four metrics are 95.5%, 94.6%, 95.0%, and 93.7%, respectively, indicating a high quality of directive sentence identification.

*4.1.2 Atomic Formula Extraction.* The quality of the atomic formula extraction is reflected by the quality of named entity recognition of arguments.

We evaluated the quality of named entity recognition with the 11,019 sentences and 16,972 arguments annotated for the training of the sequence tagging model. For each named entity type (*e.g.*, *Equal-1*, *Equal-2*, *Throw-1*, *Throw-2*), we randomly selected 80% of the annotated sentences that contain the named entity type as the training set, 10% as the validation set for hyper parameter tuning, and the other 10% as the test set. The precision, recall, and F1-measure of named entity recognition are 86.3%, 82.4%, and 84.3%, respectively, indicating high quality of the atomic formula extraction. Among the 13.7% named entities that are incorrectly recognized, 72.7% are arguments that are given wrong entity types and the other 27.3% are not arguments. We find that the quality of named entity recognition varies greatly between different types of named entities. For popular entity types that have enough training data, the quality is high, while for others that have little training data, the quality is low. For example, the precision, recall, and F1-measure for *Equal-1* and *Equal-2* are all above 85.0%. In contrast, the precision, recall, and F1-measure for *Deprecated* are all lower than 60%. This analysis indicates that the quality of atomic formula extraction can be further improved by annotating more directive sentences.

*4.1.3 Logical Relation Recognition.* We randomly selected a set of directive sentences from the training data of the directive sentence classifier to evaluate the quality of logical relation recognition. Similar to previous studies [7, 27], we adopted a statistical sampling method [23] to ensure that the estimated population parameter is at a certain confidence level. The calculated minimum number of directive sentences to be examined is 384 for this evaluation, which ensures the estimated accuracy is in the 0.05 error margin at a 95% confidence level. For the 384 directive sentences sampled for evaluation, we asked two Master students (not affiliated with this work), who are familiar with Java, to independently annotate whether the recognized clauses and logical relations are correct. When their annotations were different, a third Master student gave an additional judgment. As a result, the accuracy (*i.e.,* the ratio of sentences whose logical relations are correctly recognized) is 91.1%. The main cause for incorrectly recognized logical relations lies in linguistic patterns that are not defined.

*4.1.4 Argument Parsing.* The main challenge for argument parsing is the recognition of API elements. We evaluated the accuracy of API element recognition with the 246 API related arguments annotated for the API linking prediction model. We conducted a ten-fold cross validation: each time, nine folds of positive samples together with the corresponding negative samples are used for training and the other one fold of positive samples are used for testing. The average accuracy of API element recognition is 91.5%. The main causes for incorrectly recognized API elements include: correct API elements are not chosen as candidates; and general phrases like "the value" lead to irrelevant API elements being chosen. The accuracy of API element recognition can be further improved by better candidate selection strategies, more features, and more training data.

## 4.2 Accuracy of FOL Directive Extraction (RQ2)

We compared LeadFOL with DRONE using the replication package[7] provided by Zhou *et al.* [32], which contains FOL directives extracted for three JDK 1.8 packages (*i.e., java.awt*, *javax.swing* and *javaFX*). We randomly selected 100 methods with at least one FOL directive extracted from their dataset and compared the FOL directives extracted by DRONE with those extracted by LeadFOL. To better align their extracted FOL directives, we transformed the results of DRONE into conjunctive normal form and split the FOL formulas into conjunction clauses for comparison. Three of the authors manually constructed a golden set for the comparison.

As a result, LeadFOL produces 220 conjunction clauses for the 100 methods with a precision of 85.0%; DRONE produces 59 clauses with a precision of 50.8%. DRONE only supports four types of parameter constraints (*i.e.,* not null, nullness allowed, range limitation, and type restriction), while LeadFOL can extract more types of directives. Among the results of LeadFOL, 127 clauses (86.6% of which are correct) include predicates that are not supported by DRONE, *e.g.,* those related to return value.

To further compare with DRONE over the four types of parameter constraints that DRONE supports, we identify 104 conjunction clauses of these types from the golden set and use them to evaluate the accuracy of two approaches. The results are shown in Table 3, where "Correct" means the clauses that are correctly extracted by the approach, "Incorrect" means the clauses that are extracted with mistakes, and "Missing" means the clauses in the golden set that are not extracted by the approach. Of the 104 conjunction clauses, LeadFOL correctly extracts 77, incorrectly extracts 16, and misses 11, resulting in a precision of 82.8% and a recall of 74.0%; DRONE correctly extracts 35, incorrectly extracts 28, and misses 41, leading to a precision of 55.6% and a recall of 33.7%.

We analyze the results and find that the results of DRONE are limited by its strict matching with linguistic patterns. For example, for the directive sentence "This must be >= 0 and < the end", the closest pattern in DRONE is "[something] must be greater/less/larger than [value]", which fails to match this sentence because DRONE does not consider the usage of ">=" and "<". Grounded on learning based techniques, LeadFOL correctly extracts the FOL directive. Another reason that LeadFOL can extract more FOL directives lies in its ability of extracting directives from method and return value descriptions. In contrast, DRONE only extracts directives from parameter and exception descriptions. For example, the sentence "If destCM is null, an appropriate ColorModel will be used" from a method description implies the directive that *destCM* (which is a

[7] https://github.com/DRONE-Proj/DRONE

**Table 3: Comparison of the FOL Directives Extracted by LeadFOL and DRONE**

| LeadFOL \ DRONE | Correct | Incorrect | Missing | Total |
|---|---|---|---|---|
| **Correct** | 27 | 25 | 25 | 77 |
| **Incorrect** | 6 | 2 | 8 | 16 |
| **Missing** | 2 | 1 | 8 | 11 |
| **Total** | 35 | 28 | 41 | 104 |

method parameter) may be null. LeadFOL can correctly extract the directive and can link "destCM" to the corresponding parameter. Moreover, DRONE often fails to recognize API elements mentioned in different forms as it relies on name matching for the recognition. For example, for the conditional clause "if input stream is null" DRONE fails to recognize "input stream" as a method parameter, while LeadFOL can link it to the corresponding parameter of type `InputStream` using the learning based technique.

## 4.3 Generalization of LeadFOL (RQ3)

To assess the performance of LeadFOL on non-JDK APIs, we randomly selected 100 API methods from Android API 27[8], and applied LeadFOL on them. As a result, we extracted 261 FOL directives for those 100 API methods.

We asked two Master students (not affiliated with this work) who are familiar with Android to independently annotate whether the extracted FOL directives are correct. When their annotations were different, a third Master student gave an additional judgment. As a result, the accuracy (*i.e.,* the ratio of FOL directives that are correctly extracted) is 81.3%. The annotators achieved a Cohen's Kappa agreement [10] of 0.868, *i.e.,* near perfect agreement.

The main reasons for errors come from different sources, including the NLP analysis tool, atomic expression extraction, and logical relation recognition. Nonetheless, we contend that the results indicate that LeadFOL can be generalized to other non-JDK libraries. Preparing additional annotation data from the specific API library may lead to an even better performance.

## 4.4 Usefulness for Supporting Code Reviews (RQ4)

Detecting API misuses is one potential application area of LeadFOL. To show the usefulness of the extracted FOL directives for detecting API misuses, we asked participants to complete code review tasks with a tool based on FOL directives and compared it with a baseline.

*4.4.1 Data Preparation.* We randomly selected API methods from JDK 1.8. For each API method, we checked the corresponding API reference documentation to ensure that it contains at least one directive sentence. Then, we searched the name of the API method on GitHub to obtain a list of candidate code snippets. We selected the first code snippet that meets the following criteria:

- the code snippet is a complete method;
- the given API method is called in the code snippet;
- the code snippet is less than 50 lines;
- there is a guard condition statement for the given API method and this guard condition statement is related to the directive sentence in the API reference documentation;

[8]https://developer.android.com/reference/packages

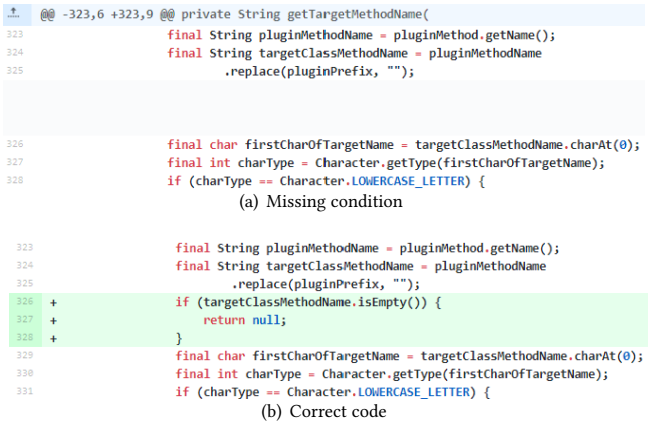


**Figure 1: API Misuse Example for Missing Guard Condition**

- the authors were able to confirm that the code snippet is bug-free.

For each code snippet found, we manually removed the guard condition statement to create a buggy code snippet variant with API misuse, because missing guard conditions are a common type of API misuse [26] in the wild. As shown in [26], 66% of the defect fixes applied in the Mozilla Firefox project are due to missing guard conditions. For example, Figure 1 shows a pull request[9] to fix a misuse of `java.lang.String.charAt(int)`, where the only difference between the buggy code and the fix is that the buggy code lacks a guard condition statement for calling `java.lang.String.charAt(int)`. We followed this scenario when creating buggy code snippets for our study. We repeated this process until we obtained 16 bug-free code snippets and their corresponding 16 buggy code snippet variants. For each code snippet, we randomly chose whether to use either the bug-free version or the buggy version in the study. As a result, we obtained 16 code snippets (8 are bug-free and 8 are buggy) and each code snippet corresponds to a code review task with ground truth. We included the bug-free code snippets in the study to avoid potential bias caused by participants knowing that all code they see in the study is buggy beforehand. All 16 tasks are available in our replication package[10].

*4.4.2 Tool Support.* The study participants were to uses two tools, assisting them during the code review task. Both tools support only JDK APIs.

We designed a tool based on the FOL directives extracted by LeadFOL. This tool accepts a code snippet as input and highlights the API methods with FOL directives extracted from API documentation. If the developer clicks an API method, it will show its FOL directives in detail. In order for participants to understand the API, the tool also shows basic information about the API, including its signature, the first sentence of its description, and the definition of its parameters. Our tool only highlights APIs with directives, not all the APIs involved.

[9]https://github.com/magento/magento2-phpstorm-plugin/pull/416/commits/d2e840d0e0d905fdb76e74386586a4ff2756f68f
[10]https://github.com/FudanSELab/Research-ESEC-FSE2021-APIDirective/blob/main/CodeReview/Code%20Review%20Tasks.xlsx

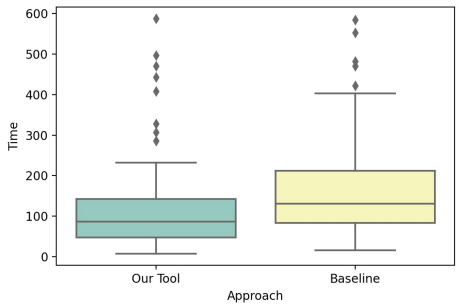


**Figure 2: Completion Time for Code Reviews**

The baseline tool is documentation-based and shows links to the API reference documentation for all APIs involved in the code snippet. When using the baseline, participants are allowed to use search engines but the search scope is limited to the JDK reference documentation. We limit participants' access to JDK reference documentation in the baseline because we want to make the two groups (*i.e.,* using our tool or the baseline ) work based on the same information source (*i.e.,* the documentation) but with different means (FOL directive extraction vs. information retrieval).

*4.4.3 Protocol.* We asked 16 Master students with 1–4 years of Java programming experience to participate in the study. They represent novice developers, which are the primary target audience for our code review tool. We conducted a pre-experiment survey on their Java programming experience and divided them into two roughly equivalent groups ($G_A$ and $G_B$) based on the survey. We randomly divided the 16 tasks into two groups ($T_A$ and $T_B$), each with 4 tasks with buggy code snippets and 4 tasks with bug-free code snippets.

We asked participants to complete code review tasks with our tool and the baseline tool. We adopted a balanced treatment distribution for the groups. Participants in group $G_A$ were asked to complete the tasks in group $T_A$ with our tool and the tasks in group $T_B$ with the baseline. Conversely, participants in group $G_B$ were asked to complete the tasks in group $T_A$ with the baseline and the tasks in group $T_B$ with our tool. For each participant, the tasks were interleaved, one completed with our tool and one with the baseline. For each code review task, participants were required to check whether the code snippet contains API misuses and explain the reasons for their judgment. The answer and completion time of each participant for each task were recorded.

*4.4.4 Result.* We checked the participants' answers for each task and evaluated their correctness through the ground truth. From a total of 256 answers, we received 20 incorrect answers (9 using our tool and 11 using the baseline), mainly because participants did not understand the code or lacked background knowledge (*e.g.,* the concept of thread lock). We removed those incorrect answers from the following analysis since our focus is on the efficiency of the code review process. We selected relatively simple tasks and we expected that the participants would be able to solve all the tasks correctly.

Figure 2 shows the participants' completion time using our tool and the baseline. Using our tool, participants completed the tasks **30.4%** faster (112s vs. 161s on average) compared to the baseline. We used Welch's T-test [29] for verifying the statistical significance of this difference. The difference in time is statistically significant ($p = 0.0006 << 0.05$). In addition, the informal feedback we received from participants shows that when using the baseline they usually spent a lot of time to search and read the API documentation to check whether the API had directives and to understand the directives. When using our tool, participants could quickly filter out APIs that do not have directive sentences, and focus on APIs with directives that were highlighted by our tool. Compared with the original sentences, FOL directives are intuitive and easy to understand.

The participants also gave us some suggestions for improvement, such as, sorting the FOL directives involved in order of importance, giving priority to important directives (*e.g.,* directives that can cause exceptions), and hoping that we can provide correct API usage code samples.

We conclude that using our tool that provides the FOL directives extracted by LeadFOL, significantly decreases the amount of time developers need for code review tasks related to API misuse.

## 4.5 Threats to Validity

Both the predicates identification and the evaluation involve data annotation and data sampling, thus common threats to the internal validity include subjective judgment of the involved participants and randomness of the sampling. For minimizing such threats, we followed commonly used sampling and data analysis techniques, involving multiple annotators, conflict resolution steps, and reported agreement coefficients, where appropriate. Since we make our data available for replication, these data sets can be further evolved and corrected (if needed) by other researchers.

A threat to the external validity is the limited number of subjects (*e.g.,* API libraries, documentation types, directive sentences) considered in the predicate identification study and in the evaluation. Our findings may not generalize to other libraries or documentation types.

# 5 RELATED WORK

API documentation is an important knowledge source for developers, and there are many studies on the knowledge in API documentation and its patterns. Robillard *et al.* [9] identified 12 knowledge patterns in API documentation, such as functionality, concepts, and directives. Among them, directives are especially important knowledge for developers because they are related to the correct usage of APIs. Monperrus *et al.* [13] focused on the directives in API documentation and identified 23 kinds of API directives. Other studies related to API documentation focus on different aspects, such as API documentation evolution [22] and documentation reuse [16].

Another line of work focused on extracting sentences providing specific types of knowledge from API documentation. Liu *et al.* [8] extracted functionality and directives sentences from API documentation to construct an API knowledge graph based on a sentence classifier. Li *et al.* [7] extracted API caveats from API documentation based on syntactic patterns to improve accessibility. That research

extracted sentences containing knowledge from API documentation without further processing. Different from those works, we not only use the directive sentence classifier to extract sentences containing directives, but also parse them into a standardized form expressed in first-order predicate logic.

Other research extracted specific kinds of formally expressed API directives from documentation, *e.g.,* resource specifications [31], temporal constraints [17], parameter constraints [25], and call-order/condition-checking [20]. Recently, Zhou *et al.* [32] proposed an approach called DRONE that extracts API directives represented in first-order logic (FOL) formulas to express four types of parameter constraints (*i.e.,* Nullness, Nullable, Range Limitation, Type Restriction). These approaches rely on linguistic patterns to extract formal expressions of API directives, hence can extract fewer directive types with lower accuracy. In contrast, our learning-based approach is more general and achieves higher extraction accuracy.

The work of Zhou *et al.* [32] is most similar to ours. Next we will make a deeper comparison. First, they supports four types of parameter constraints, while we support additional usage-related and return-value-related directives. Second, they relies on 64 manually defined linguistic patterns to extract FOL formulas and parse arguments based on name match. In contrast, we design a 5-step pipeline using (deep) learning technologies to extract FOL formulas and each step is designed as a subtask with clear input/output definition. Among the steps, the atomic formula extraction uses a joint-learning model which can more precisely identify arguments and predicates. The argument parsing step uses an API graph and combines several different similarity metrics to parse the API elements references. Therefore, our approach is more general and can extract more directives, more accurately.

## 6 CONCLUSIONS AND FUTURE WORK

We analyzed a large set of API directives extracted from the JDK 1.8 API documentation. We identified 24 predicates, classified in four categories, and three types of arguments that are used for representing API directives in first order logic (FOL) format.

Based on these findings, we proposed a learning based approach for extracting FOL representations of API directives, called LeadFOL. Our intrinsic and extrinsic evaluation shows that LeadFOL can accurately extract more FOL directives than a state-of-the-art approach (*i.e.,* DRONE [32]) and that the extracted FOL directives help developers detect of API misuse problems faster, during code reviews.

Our future work will focus on improving the accuracy of FOL directive extraction and using the extracted FOL directives for different purposes, such as question answering.

## 7 DATA AVAILABILITY

All the data used in this study is provided in the replication package [2].

## ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China under Grant No. 61972098.

## REFERENCES

[1] 2021. *gensim.* Retrieved February 25, 2021 from https://radimrehurek.com/gensim/

[2] 2021. *Replication Package.* Retrieved June 15, 2021 from https://fudanselab.github.io/Research-ESEC-FSE2021-APIDirective/

[3] 2021. *word2vec-api.* Retrieved February 25, 2021 from https://github.com/3Top/word2vec-api

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers).* 4171–4186. https://doi.org/10.18653/v1/n19-1423

[5] Davide Fucci, Alireza Mollaalizadehbahnemiri, and Walid Maalej. 2019. On using machine learning to identify knowledge in API reference documentation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.* 109–119. https://doi.org/10.1145/3338906.3338943

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[7] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving API Caveats Accessibility by Mining API Caveats Knowledge Graph. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018.* 183–193. https://doi.org/10.1109/ICSME.2018.00028

[8] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating query-specific class API summaries. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.* 120–130. https://doi.org/10.1145/3338906.3338971

[9] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Trans. Software Eng.* 39, 9 (2013), 1264–1282. https://doi.org/10.1109/TSE.2013.12

[10] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica* 22, 3 (2012), 276–282.

[11] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (Eds.). 3111–3119.

[12] Frederic P Miller, Agnes F Vandome, and John McBrewster. 2009. Levenshtein distance: Information theory, computer science, string (computer science), string metric, damerau? Levenshtein distance, spell checker, hamming distance. (2009).

[13] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17, 6 (2012), 703–737. https://doi.org/10.1007/s10664-011-9186-4

[14] Hiroki Nakayama, Takahiro Kubo, Junya Kamura, Yasufumi Taniguchi, and Xu Liang. 2018. doccano: Text Annotation Tool for Human. https://github.com/doccano/doccano Software available from https://github.com/doccano/doccano.

[15] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, Vol. 1. 380–384.

[16] Mohamed A Oumaziz, Alan Charpentier, Jean-Rémy Falleri, and Xavier Blanc. 2017. Documentation reuse: Hot or not? An empirical study. In *International Conference on Software Reuse.* Springer, 12–27. https://doi.org/10.1007/978-3-319-56856-0_2

[17] Rahul Pandita, Kunal Taneja, Laurie Williams, and Teresa Tung. 2016. ICON: Inferring Temporal Constraints from Natural Language API Descriptions. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016.* IEEE Computer Society, 378–388. https://doi.org/10.1109/ICSME.2016.59

[18] Lev-Arie Ratinov and Dan Roth. 2009. Design Challenges and Misconceptions in Named Entity Recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning, CoNLL 2009, Boulder, Colorado, USA, June 4-5, 2009.* 147–155.

[19] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 3980–3990. https://doi.org/10.18653/v1/D19-1410

[20] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 461–472. https://doi.org/10.1145/3324884.3416551

[21] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empir. Softw. Eng.* 16, 6 (2011), 703–732.

[22] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. 2011. An empirical study on evolution of API documentation. In *International Conference on Fundamental Approaches To Software Engineering*. Springer, 416–431. https://doi.org/10.1007/978-3-642-19811-3_29

[23] Ravindra Singh and Naurang Singh Mangat. 2013. *Elements of survey sampling*. Vol. 15. Springer Science & Business Media.

[24] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2019. Investigating next steps in static API-misuse detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 265–275. https://doi.org/10.1109/MSR.2019.00053

[25] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche (Eds.). IEEE Computer Society, 260–269. https://doi.org/10.1109/ICST.2012.106

[26] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining Alternative Patterns for Detecting Neglected Conditions. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 283–294. https://doi.org/10.1109/ASE.2009.72

[27] Chong Wang, Xin Peng, Mingwei Liu, Zhenchang Xing, Xuefang Bai, Bing Xie, and Tuo Wang. [n.d.]. A learning-based approach for automatic construction of domain glossary from source code and documentation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). 97–108. https://doi.org/10.1145/3338906.3338963

[28] Jason W. Wei and Kai Zou. 2019. EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 6381–6387. https://doi.org/10.18653/v1/D19-1670

[29] Bernard L Welch. 1947. The generalization of Student's problem when several different population variances are involved. *Biometrika* 34, 1/2 (1947), 28–35. https://ci.nii.ac.jp/naid/10026469617/en/

[30] Suncong Zheng, Feng Wang, Hongyun Bao, Yuexing Hao, Peng Zhou, and Bo Xu. [n.d.]. Joint Extraction of Entities and Relations Based on a Novel Tagging Scheme. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. 1227–1236. https://doi.org/10.18653/v1/P17-1113

[31] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring Resource Specifications from Natural Language API Documentation. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 307–318. https://doi.org/10.1109/ASE.2009.94

[32] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald C Gall. 2018. Automatic detection and repair recommendation of directive defects in Java API documentation. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2872971