

ICON: Inferring Temporal Constraints from Natural Language API Descriptions

Rahul Pandita*, Kunal Taneja†, Tao Xie‡, Laurie Williams*, Teresa Tung†

*Department of Computer Science, North Carolina State University, Raleigh, NC, USA

†Accenture Technology Labs, San Jose, CA, USA

‡Department of Computer Science, University of Illinois, Urbana-Champaign, IL, USA

rpandit@ncsu.edu, k.a.taneja@accenture.com, taoxie@illinois.edu, williams@csc.ncsu.edu, teresa.tung@accenture.com

Abstract—Formal specifications of using a libraries Application Programming Interface (API) can be used in conjunction with formal analysis tools such as model checkers and runtime verifiers to assess the quality of a software. Despite being desirable, most API’s do not have formal specifications. In contrast, API documents contain valuable information about the usage in natural language. However, formal analysis tools are not designed to work on specifications in natural languages. Manually writing formal specifications based on natural language text in API documents can be prohibitively time consuming and error prone. To address this issue, we propose a natural language processing based automated approach ICON, to infer formal specifications from natural language text of API documents. In particular, we focus on temporal constraints, that are defined as constraints on the *allowed sequence of invocations of methods within an API*. To evaluate our proposed approach, we applied ICON to infer temporal constraints from commonly used package `java.io` from JDK API and Amazon S3 REST API. Our evaluation results show that ICON achieves an average of 60% precision and 80% recall in inferring TT temporal constraints from more than 2400 sentences of API documents.

I. INTRODUCTION

Temporal specifications [2] of an Application Programming Interface (API) describe allowed sequences of invocations of methods within an API. These specifications govern the secure and robust operation of client software using these APIs. If these specifications are formal (machine-readable such as linear temporal logic), they can be used as a basis of formal analysis tools such as model checker and runtime verifiers in detecting the violations of these temporal constraints as defects.

Despite being desirable, most API’s do not have formal specifications. In contrast, API developers commonly describe correct temporal usage in natural language text in API documents. Typically, such documents are provided to client-code developers through online access, or are shipped with the API code. Typically, an API document describes both the constraints on that method parameters as well as the temporal constraints in terms of the methods must be invoked pre/post the current method. We observed that roughly 12% of the sentences describing some sort of specification constraints in Amazon S3 REST API documentations describe temporal constraints. Although API documents contain temporal constraints in natural language, formal analysis tools

are not designed to work on specifications written in natural languages.

One way of addressing the issue, is to manually convert the natural language API description into formal specifications. However, manually writing formal specifications based on natural language text in API documents can be prohibitively time consuming and error prone [28], [40]. For instance, Wu et al. [40] report that it took one of the authors more than 10 hours to browse the documentation of one method of a web Service API, even before they attempted to formalize the constraints on the method.

We propose a novel approach to infer temporal usage constraints from natural language text of API documents. We propose a new approach that apply Natural Language Processing (NLP) on natural language text API documents to automatically infer temporal usage constraints. A generic temporal relation is defined as an *interpropositional relation that communicates the simultaneity or ordering in time of events or states*. In terms of API method invocations, we interpret temporal relationships as ‘*the allowed sequence of invocations of methods*’. For example, consider the following statement in Amazon S3 REST API “You must initiate a multipart upload before you can upload any part.” This statement can be interpreted as multipart-upload method must be invoked before invoking any part-upload method in the API.

There are existing approaches focus on inferring just the method pre-post conditions in terms of parameter constraints either using program analysis techniques [5], [11], [13]–[15] or using NLP [25], [40]. In contrast, temporal constraints are beyond these parameter constraints. Temporal constraints, in general focus on rules related to orchestration of methods within an API rather than focusing on the requirements on the input parameters of these methods. Our proposed approach addresses the following challenges to infer temporal constraints automatically from API documents.

First, relying just on type information will result in a incomplete list of temporal constraints. Although, in typed languages some temporal constraints are enforced by type system. For instances a method (m) accepting input parameter (i) of type (t) mandates that (at least one) method (m') be invoked whose return value is of type (t). However, temporal constraints are not limited by the type definitions

(i.e., requirements on the type of input parameters), and are currently expressed mostly in natural language in API documents, as described in the example previously. We propose to address the issue by augmenting the type definitions based temporal constraints with the constraints inferred from the natural language text in API documents.

Second, inferring temporal constraints from the natural language text itself is challenging. There are existing challenges in NLP for software engineering domain namely *ambiguity*, *programming keywords*, and *semantic equivalence* identified in previous work [25]. In addition to these challenges, we have an added challenge of identifying the method referenced in the natural language text to infer temporal constraint. Recall, the description sentence “You must initiate a multipart upload before you can upload any part.” In this sentence, phrases “multipart upload” and “upload any part” refer to individual methods in the API. Identifying these phrases as method invocation instances require domain dictionaries. Ad-hoc constructing these dictionaries is prohibitively time and resources intensive. To address this challenge, we propose to build domain-dictionaries systematically from API documents themselves.

In summary, the proposed work leverages natural language description of API’s to infer temporal constraints of method invocations. As the proposed work analyzes API documents in natural language, it can be reused independent of the programming language of the library. Additionally, our approach complements existing mining based approaches [4], [37], [39], [42] that partially address the problem by mining for common usage patterns among client software that use the API. The proposed work in general, makes the following contributions:

- A NLP based approach that infers temporal constraints of method invocations. To the best of our knowledge, our approach is the first one to apply NLP for the goal of inferring temporal specifications from API documents.
- A prototype implementation of our approach based on extending the Stanford Parser [16], [32], which is a natural language parser to derive the grammatical structure of sentences. An open source implementation of the prototype is publicly available on our project website¹.
- An evaluation of proposed approach on Java 7 and Amazon S3 REST API.

The rest of the paper is organized as follows. Section II presents an real world examples that motivate our approach. Section III discusses related work in this area. Section IV presents the background on code contracts as well as NLP. Section V presents our approach. Section VI presents evaluation of our approach. Section VII presents a brief discussion and future work. Finally, Section VIII concludes.

¹<https://sites.google.com/site/icon>

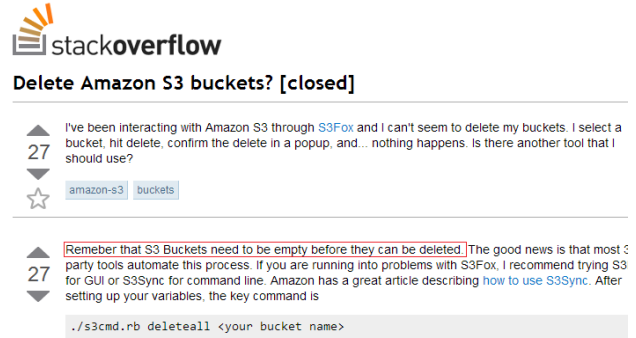


Figure 1. The Query posted on Stackoverflow form regrading Amazon S3 REST API

II. MOTIVATING EXAMPLE

We now present some real world examples to motivate our approach. In particular, through these examples we demonstrate that developers often overlook the documentation. Thus, the resultant system is inconsistent with the intentions that a developer had in mind initially. This inconsistency oftentimes result in undesirable consequences ranging from developer frustration, customer dissatisfaction, to actual financial losses.

For example consider the question asked in *Stack Overflow*² as shown in Figure 1. Stack Overflow is a question and answer site for professional and enthusiast programmers. The query is about the delete functionality of a third-party software S3Fox to interact with Amazon S3 REST API. In particular, the query complains about an issue in delete bucket functionality of the S3Fox. In particular, the issue was because the S3Fox developers overlooked the specifications in Amazon S3 REST API developer documentation. Figure 2 shows the documentation of delete bucket method. The documents clearly states (outlined in red for clarity) that before deleting the bucket the objects in the buckets must be deleted. Although the issue was fixed but notice that the response on the *Stack Overflow* encouraged the person asking query to switch to another product, thus potentially resulting in a loss of revenue attributed to customer dissatisfaction.

In yet another example a developer points out the financial losses he suffered because he was incorrectly using the previously described delete bucket functionality as shown in Figure 3. The problems in the previously described examples could have been alleviated if developers paid more attention to the documentation. However, as pointed out by Novick et. al [23] that developers often overlook the documentation. Furthermore, since these documents are written in natural language they cannot be subjected to machine based verification. Thus, there is need for an approach to translate the constraints described in natural language into a more formal notation. Although, we only use examples from

²<http://stackoverflow.com/>

DELETE Bucket

Description

This implementation of the DELETE operation deletes the bucket named in the URI. All objects (including all object versions and Delete Markers) in the bucket must be deleted before the bucket itself can be deleted.

Requests

Syntax

```
DELETE / HTTP/1.1
Host: BucketName.s3.amazonaws.com
Date: date
Authorization: signatureValue
```

Request Parameters

This implementation of the operation does not use request parameters.

Request Headers

This implementation of the operation uses only request headers that are common to all operations. For more information, see [Common Request Headers](#).

Request Elements

This implementation of the operation does not use request elements.

Figure 2. The online API document for DELETE Bucket operation in Amazon S3 REST API

How to Delete Large Buckets in Amazon S3

2-01-2010 | AMAZON WEB SERVICES

While reviewing our Amazon S3 account earlier this week, I came across an old unused bucket that had apparently sat dormant for an embarrassingly long time. As great as cloud hosting may be, paying for excess stuff you don't use or need is simply idiotic - hence my determination to rid this problem in 5 minutes.

Unfortunately, I found out S3 buckets cannot be deleted when there are files inside it. Also, if you have a few hundred thousand files in a bucket, trying to list those files will crash most if not all UI based S3 tools that have a limit of listing only 1,000 files at one time.

Figure 3. The experience article posted by a developer regarding Amazon S3 REST API

the developer forums and blogs for Amazon S3 REST API developer documentation to motivate our approach. In next section, we briefly discuss the related work in this area.

III. RELATED WORK

Our proposed approach touches a few research areas such as software verification, NLP on software engineering artifacts, and document augmentation. We next discuss the relevant work pertinent to our proposed approach in these areas.

A. Code Contracts - Formal Specifications

Design by contracts has been an influential concept in the area of software engineering in the past decade. A significant amount of work has been done in automated inference of code contracts. There are existing approaches that statically or dynamically extract code contracts [6], [22], [38]. However, a combination of developer written and automatically inferred contracts seems to be the most effective approach [11], [26]. Furthermore, there are existing approaches that infer code-contract-like specifications (such as behavioral model, algebraic specifications, and exception specifications) either dynamically [13]–[15] or statically [5], [11] from source code and binaries. In contrast, the approach presented in this work infers specifications from the natural language text in API documents, thus complementing these existing approaches when the source code or binaries of the API library is not available.

B. NLP in Software Engineering

NLP techniques are increasingly applied in the software engineering domain. NLP techniques have been shown to be useful in requirements engineering [12], [29], [30], usability of API documents [9], and other areas [20], [24], [44]. We next describe the most relevant approaches.

1. *Access Control Policies*: Xiao et al. [41] and Slankas et al. [31] use shallow parsing techniques to infer Access Control Policy (ACP) rules from natural language text in use cases. The use of shallow parsing techniques works well on natural language texts in use cases, owing to well formed structure of sentences in use case descriptions. In contrast, often the sentences in API documents are not well formed. Additionally, these approaches do not deal with programming keywords or identifiers, which are often mixed within the method descriptions in API documents.

2. *Resource Specifications*: Zhong et al. [43] employ NLP and Machine Learning (ML) techniques to infer resource specifications from API documents. Their approach uses machine learning to automatically classify such rules. In contrast, we attempt to parse sentences based on semantic templates and demonstrate that such an approach performs reasonably well. Furthermore, the performance of the proposed approach is dependent on the quality of the training sets used for ML. In contrast, approach presented in this work is independent of such training set and thus can be easily extended to target respective problems addressed by them.

3. *Code Comments*: Tan et al. [33] and Hwei-Tan et al. [35] applied an NLP and ML based approach on code comments/ Javadoc comments to detect mismatches between these comments and implementations. They rely on pre-defined rule templates targeted towards method invocation and lock related comments, thus limiting their scope both in terms of application area as well as language used in the comments. In contrast, approach presented in this report relies on generic natural language based templates thus relaxing the restriction on the style of the language used to describe specifications.

4. *API Description*: Most closely related work to the approach presented here is Pandita et al. [25] work on inferring parameter constraints from method descriptions in the API documents. In contrast this approach deals with the temporal constraints. The approach presented here is a significant extension to the infrastructure used in the previous work as documented in Section V.

C. Augmented Documentation

Another related field has been that of improving the documentation related to an software API [9], [34]. We next describe most relevant approaches. Dekel et al. [9], were the first to create a tool namely eMoose, an Eclipse³ based

³<http://www.eclipse.org/>

plug-in that allowed developers to create directives (way of marking the specification sentences) in the default API documentation. These directives are highlighted whenever they are displayed in the eclipse environment. Lee et al. [18] improved upon their work by providing a formalism to the directives proposed by Dekel et al. [9], thus allowing tool based verification. However, a developer has to manually annotate such directives. In contrast, our proposed approach both identifies the sentences pertaining to temporal constraints and infers the temporal constraints automatically.

In next section, we briefly introduce the NLP techniques used by our approach.

IV. BACKGROUND

Natural language is well suited for human communication, but converting natural language into unambiguous specifications that can be processed and understood by computers is difficult. However, research advances [7], [8], [16], [17] have increased the accuracy of existing NLP techniques to annotate the grammatical structure of a sentence. These advances in NLP have inspired researchers/practitioners [24], [25], [31], [36], [41] to adapt/apply NLP techniques to solve problems in software engineering domain.

In particular, this work proposes novel techniques on top of previously proposed techniques [24], [25] to demonstrate the effectiveness of applying NLP on API documents. We next briefly introduce the techniques used in this work that have been grouped into broad categories. We first introduce the core NLP techniques used in this work. We then introduce the software engineering specific NLP techniques proposed in previous work [24], [25] that have been used in this work.

A. Core NLP techniques

Parts Of Speech (POS) tagging [16], [17]. Also known as ‘word tagging’, ‘grammatical tagging’ and ‘word-sense disambiguation’, these techniques aim to identify the part of speech (such as noun, verbs, etc.), a particular word in a sentence belongs to. The most commonly used technique is to train a classification parser over a previously known data set. Current state of the art approaches have been demonstrated to achieve 97% [32] accuracy in classifying POS tags for well written news articles.

Phrase and Clause Parsing. Also known as chunking, this technique divides a sentence into a constituent set of words (or phrases) that logically belong together (such as a Noun Phrase and Verb Phrase). Chunking thus further enhances the syntax of a sentence on top of POS tagging. Current state-of-the-art approaches can achieve around 90% [32] accuracy in classifying phrases and clauses over well written news articles.

Typed Dependencies [7], [8]. The Stanford typed dependencies representation is designed to provide a simple description of grammatical relationships directed towards

non-linguistics experts to perform NLP related tasks. It provides a hierarchical structure for the dependencies with precise definitions of what each dependency means, thus facilitating machine based manipulation of natural language text.

Named Entity Recognition [10]. Also known as ‘entity identification’ and ‘entity extraction’, these techniques are a subtask of IE that aims to classify words in a sentence into predefined categories such as names, quantities, expression of times, etc. These techniques help in associating predefined semantic meaning to a word or a group of words (phrase), thus facilitating semantic processing of named entities.

B. Software engineering specific NLP techniques

Noun Boosting [25]. Accurate annotation of POS tags in a sentence is fundamental to effectiveness of any advanced NLP technique. However, as mentioned earlier, POS tagging work well on well written news articles which does not necessarily entail that they work well with domain specific text as well. Thus, noun boosting is a necessary precursor to application of POS tagging on domain specific text. In particular, with respect to API documents certain words have a very different semantic meaning, in contrast to general linguistics that causes incorrect annotation of POS tags.

Consider the word `POST` for instance. The online Oxford dictionary ⁴ has 8 different definition of word `POST`, and none of them describes `POST` as a HTTP method ⁵ supported by REST API. Thus existing POS tagging techniques fail to accurately annotate the POS tags of the sentences involving word `POST`.

Noun Boosting identifies such words from the sentences based on a domain-specific dictionaries, and annotates them appropriately. The annotation assists the POS tagger to accurately annotate the POS tags of the words thus inturn increasing accuracy of advanced NLP techniques such as chunking and typed dependency annotation.

Lexical Token Reduction [24]. These are a group of generic preprocessing heuristics to further improve the accuracy of core NLP techniques. The accuracy of core NLP techniques is inversely proportional to the number of lexical tokens in a sentence. Thus, the reduction in the number of lexical tokens greatly increases the accuracy of core NLP techniques. In particular, following heuristics have been used in previous work to achieve the desired reduction of lexical tokens:

- **Period Handling.** Besides marking the end of a sentence in simplistic English, the character period (‘.’) has other legal usages as well such as decimal representation (periods between numbers). Although legal,

⁴http://oxforddictionaries.com/us/definition/american_english/post?q=POST

⁵In HTTP vocabulary `POST` means: “Creates a new entry in the collection. The new entry’s URI is assigned automatically and is usually returned by the operation”

such usage hinder detection of sentence boundaries, thus causing core NLP techniques to return incorrect or imprecise results. The text is pre-processed by annotating these usages for accurate detection of sentence boundaries.

- **Named Entity Handling.** Sometimes a sequence of words correspond to the name of entities that have a specific meaning collectively. For instance, consider the phrases “Amazon S3”, “Amazon simple storage service”, which are the names of the service. Further resolution of these phrases using grammatical syntax is unnecessary and would not bring forth any semantic value. Also these phrases contribute to length of a sentence that in turn negatively affects the accuracy of core NLP techniques. This heuristic annotates the phrase representing the name of the entities as a single lexical token.
- **Abbreviation Handling.** Natural-language sentences often consist of abbreviations mixed with text. This can result in subsequent components to incorrectly parse a sentence. This heuristic finds such instances and annotates them as a single lexical unit. For example, text followed by abbreviations such as “Access Control Lists (ACL)” is treated as single lexical unit. Detecting such abbreviations is achieved by using the common structure of abbreviations and encoding such structures into regular expressions. Typically, regular expressions provide a reasonable approximation for handling abbreviations.

Intermediate-Representation Generation [24]. This technique accepts the syntax-annotated sentences and builds a First-Order-Logic(FOL) representation of the sentence. Earlier researches have shown the adequacy using FOL for NLP related analysis tasks [25], [29], [30]. In particular, WHYPER [24] demonstrates the effectiveness of this technique, by constructing an intermediate representation generator based on shallow parsing [3] techniques. The shallow parser itself is implemented as sequence of cascading finite state machines based on the functions of stanford-typed dependencies [7], [8], [16], [17].

In next section we describe our proposed generic approach to infer constraints from API documents.

V. ICON DESIGN

We next present our framework for inferring specifications from the method descriptions in API Documents. Figure 4 gives an overview of our framework. Our framework consists of four major components: an API parser, a preprocessor, a text-analysis engine, and a postprocessor.

The API parser accepts online API description and extracts method descriptions. The pre-processor accepts method descriptions and preprocesses the sentences in the descriptions, such as annotating sentence boundaries and reducing lexical tokens. The text-analysis engine accepts

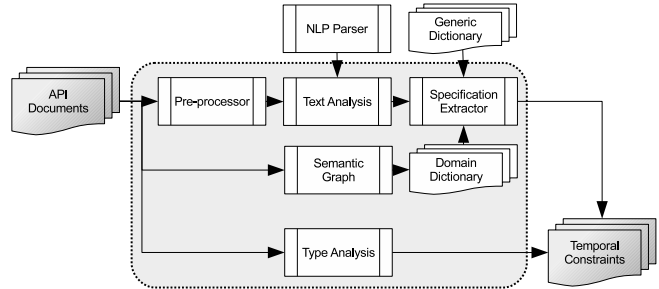


Figure 4. Overview of ICON framework

the pre-processed sentences and annotates them using an NLP parser. The text-analysis engine further transforms the annotated sentences into the first-order-logic (FOL) representation and then leverages the FOL representation of a sentence to infer specifications. Finally, the postprocessor undoes any modifications done by the preprocessor to get refined specifications.

Besides the previously described major components the framework also consists of an external component, namely target specification generator. This component accepts the specification generated by our framework and using mapping functions to a target specification language generates specifications in the target language of choice. We next describe each component in detail.

A. API Parser

Our API parser accepts the online API documents and extracts method descriptions from it. In particular, our parser extracts the following fields within method descriptions: 1) *Summary of the API method*, 2) *Summary of request elements of the API method*, and 3) *Summary of response elements of the method (inclusive of error responses)*.

This step is required to extract the desired descriptive text from the various presentation styles of the API documents. In particular, different API documents may have different styles of presenting information to developers. This difference in style may include the difference in the level of detail presented to the developer. We thus chose very only generic fields that are essential to API methods.

Currently our prototype implementation extracts this information from online Amazon S3 REST API developer documentation only. However, almost all of the REST API developer documents are provided online as structured web-pages. Thus, current implementation of parser can be easily extended to extract desired information from any REST API developer documents.

B. Preprocessor

The preprocessor accepts API method descriptions extracted by the parser and preprocesses the natural language text, to be further analyzed by the text-analysis engine. The preprocessing steps are required to increase the accuracy

core NLP techniques (described in Section IV-A) that are used in the subsequent phases of ICON framework. In particular, the preprocessor first employs the noun boosting followed by heuristics listed under lexical token reduction, as introduced in Section IV-B.

From an implementation perspective, we manually built the domain dictionaries for the preprocessing using the glossary of terms collected from the websites pertaining to REST API. We further leveraged the HTML style information in Amazon S3 REST API developer documentation to look for words that were highlighted in code like format. We further leveraged WordNet to maintain a static lookup table of shorthand words to aid named entity handling and abbreviation handling.

Although the previous techniques and heuristics significantly lower the number of lexical tokens in a sentence, some sentences may still contain a considerable number of lexical tokens to overwhelm the POS tagger. To address this issue, we propose a novel technique (*‘Frequent Phrases Reduction’*) to further reduce the number of lexical tokens in a sentence by annotating frequent phrases as a single lexical unit.

In particular, we use n-gram based approach as means to achieve this reduction. In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n words from a given sequence of text or speech. We first calculate the most frequently occurring n-grams in the text body. In particular, we are interested in the n-grams of length 4 or greater to achieve a reasonable reduction. We then prune the list of n-grams based on a subsumption. We consider a n-gram of length k (n_k) to subsume n-gram of length k-1 (n_{k-1}) iff n_{k-1} is a substring of (n_k) and the frequency of occurrence of n_{k-1} equals frequency of occurrence of n_k . Finally, we rank the list of n-grams based on the frequency of their occurrence in the text, and select top-k n-grams for reduction.

From an implementation perspective we used Apache Lucene[®] [21] to achieve n-gram reduction. Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.

C. NLP Parser

The NLP parser accepts the pre-processed documents and annotates every sentence within each document using core NLP techniques described in Section IV-A. From an implementation perspective, we chose the Stanford parser [?]. However, this component can be implemented using any other existing NLP libraries or frameworks. In particular, we annotate each sentence with POS tags, named-entity Annotations and Stanford-typed dependencies. For more details on these techniques and their application, please refer to [7], [8], [24], [25], [36].

D. Text Analysis Engine

The text analysis engine component accepts the annotated documents and creates an intermediate representation of each sentence. We define our representation as a tree structure that is essentially a First-Order-Logic (FOL) expression. Research literature provides evidence of the adequacy of using FOL for NLP related analysis tasks [24], [25], [29], [30].

In our representation, every node in the tree except for the leaf nodes is a predicate node. The leaf nodes represent the entities. The children of the predicate nodes are the participating entities in the relationship represented by the predicate. The first or the only child of a predicate node is the governing entity and the second child is the dependent entity. Together the governing entity, predicate and the dependent entity node form a tuple.

As described in Section IV-B the intermediate representation generation technique is based on the principle of shallow parsing [3]. In particular, the intermediate-representation technique is implemented as a function of Stanford-typed dependencies [?], [7], [8], [17], to leverage the semantic information encoded in Stanford-typed dependencies.

However, we observed that such implementation is overwhelmed by complex sentences. This limitation mandates the use of additional novel technique of *‘Frequent Phrases Reduction’* in preprocessing phase. We further improve the accuracy of intermediate-representation generation by proposing an hybrid approach, i.e. taking into consideration both the POS tags as well as Stanford-typed dependencies. The POS tags which annotate the syntactical structure of a sentence are used to further simplify the constituent elements in a sentence. We then use the Stanford-typed dependencies that annotate the grammatical relationships between words to construct our FOL representation. Thus, the intermediate representation generator used in this work is two phase process as opposed to previous work [24], [25]. We next describe these two phases:

POS Tags: We first parse a sentence based on the function of POS tags. In particular, we use semantic templates to logically break a sentences into smaller constituent sentences. For instance, consider the sentence:

“All objects (including all object versions and Delete Markers) in the bucket must be deleted before the bucket itself can be deleted.”.

The Stanford parser inaccurately annotates the Stanford-typed dependencies of the sentence because of presence of different clauses acting on different subject-object pairs. We thus break down the sentence into two smaller tractable sentences:

“All objects in the bucket must be deleted before the bucket itself can be deleted.”

“All objects including all object versions and Delete Markers.”

Table I shows a list the semantic templates used in this phase. Column “Template” describes conditions where the

template is applicable and Column “Summary” describes the action taken by our shallow parser when the template is applicable. All of these semantic templates are publicly available on our project website [1]. With respect to the previous example the template no. 3 (*A noun phrase followed by another noun/pronoun/verb phrase in brackets*) is applicable. Thus our shallow parser breaks the sentence into two individual sentences.

Stanford-typed Dependencies: This phase is equivalent to the intermediate-representation technique described in Section IV-B.

E. Specification Extractor

This component accepts the FOL representation of the sentence.

- Extracts Specifications from FOL representation.
- Specifically looking for modal modifies “can, could, may, must, should”
- Takes into consideration the negative quantifiers.
- Algorithm to infer the sequence of operations.

F. Postprocessor

undoing pre-processing and Hybrid intermediate representation generation.

G. Semantic-Graph Generator

A key way of identifying reference to a method within the API in our proposed framework is the employment of a semantic graph of an API. In particular, we propose to initially infer such graphs from API documents. Manually creating a semantic graph is prohibitively time consuming and may be error prone. We thus employ a systematic methodology (proposed previously in [24]) to infer such semantic graphs from API documents that can potentially be automated. We first consider the name of the class. We then find the synonyms terms used refer to the class in question. The synonym terms are listed as by breaking down the camel-case notation in the class name. This list is further augmented by listing the name of the parent classes and implemented interfaces if any.

We then systematically inspect the member methods to identify actions applicable to the objects represented by the class. From the name of a public method (describing a possible action on the object), we extract verb phrases. The verb phrases are used as the associated actions applicable on the object. For instance, `BufferedReader` defines operations available, close, mark, and so on. We associate these operations with the objects of type `BufferedReader`. Figure 5 shows a sub-graph of graph for `BufferedReader` class. The phrases in solid rectangles are synonyms of the class name `BufferedReader`. The phrases in rounded rectangle are the actions applicable on `BufferedReader` class.

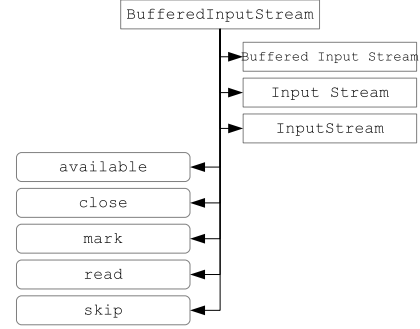


Figure 5. Semantic Graph for the `BufferedInputStream` class in Java

Algorithm 1 Action_Extractor

Input: `K_Graph g, FOL_rep rep`
Output: `String action`

```

1: String action =  $\phi$ 
2: List r_name_list = g.resource_Names
3: FOL_rep r' = rep.findLeafContaining(r_name_list)
4: List actionList = g.actionList
5: while (r'.hasParent) do
6:   if actionList.contains(r'.parent.predicate) then
7:     action = actionList.matching(r'.parent.predicate)
8:     break
9:   else
10:    if actionList.contains(r'.leftSibling.predicate) then
11:      action = actionList.matching(r'.leftSibling.predicate)
12:      break
13:    end if
14:  end if
15:  r' = r'.parent
16: end while
17: return action
```

Algorithm 2 Type_Sequence_Builder

Input: `List methodList`
Output: `Graph seq_Graph`

```

1: Graph seq_Graph =  $\phi$ 
2: Map idx = createIdx(methodList)
3: for all Method mtd in methodList do
4:   seq_Graph.addVertex(mtd)
5: end for
6: for all Method mtd in methodList do
7:   if mtd.isPublic() then
8:     if !mtd.isStatic() then
9:       List preList = idx.query(mtd.declaringType)
10:      for all Method mtd' in preList do
11:        seq_Graph.addEdge(mtd', mtd)
12:      end for
13:    end if
14:    for all Parameter param in mtd.getParameters() do
15:      if !isBasicType(param.Type) then
16:        List preList = idx.query(param.Type)
17:        for all Method mtd' in preList do
18:          seq_Graph.addEdge(mtd', mtd)
19:        end for
20:      end if
21:    end for
22:  end if
23: end for
24: return seq_Graph
```

VI. EVALUATION

We conducted an evaluation to assess the effectiveness of our approach. In our evaluation, we address three main research questions:

Table I
SEMANTIC TEMPLATES

S No.	Template	Summary
1.	Two sentences joined by a conjunction	Sentence is broken down into two individual sentences with the conjunction term serving as the connector between two.
2.	Two sentences joined by a “;”	Sentence is broken down to individual independent sentences
3.	A noun phrase followed by another noun/pronoun/verb phrase in brackets	Two individual sentences are formed. The first sentence is the same as the parent sentence sans the noun/pronoun.verb phrase in bracket. The second sentence constitutes of the noun phrase followed by noun/pronoun/verb phrase without the brackets.
4.	A noun phrase by a conditional phrase in brackets	Two individual sentences are formed. The first sentence is the same as the parent sentence sans the conditional phrase in bracket. The second sentence constitutes of noun phrases followed by conditional in the bracket.
5.	A conditional phrase followed by a sentence	Two dependent sentences are formed. The first sentence constitutes the conditional phrase. The second sentence constitutes rest of the sentence.
6.	A sentence in which the parent verb phrase is over two child verb phrases joined by a conjunction	Two dependent sentences are formed where the dependency is the conjunction. The first sentence is formulated by removing conjunction and second child verb phrase. The second sentence is formulated by removing conjunction and first child verb phrase.

- **RQ1:** What are the precision and recall of ICON in inferring usage constraints?
- **RQ2:** How does our approach fares in comparison to previous NLP approaches?

A. Subjects

We used the API documents of the following two libraries as subjects for our evaluation.

Amazon S3 REST API developer documentation provides a simple REST based web services interface that can be used to store and retrieve data on the web. Furthermore, Amazon S3 also empowers a developer with rich set of API methods to access a highly scalable, reliable, secure, fast, inexpensive infrastructure. **may be a little detail of usage statistics etc....**

TBD.

B. Summary

C. Threats to Validity

Threats to external validity primarily include the degree to which the subject documents used in our evaluations are representative of true practice. To minimize the threat, we used API documents of two representative commercial REST API: one dealing with online storage and the other **TBD**. The Amazon S3 REST API developer documentation documents describe one of the most popularly used and online storage APIs. We also used the **TBD**. Furthermore, the difference in the functionalities provided by the two projects also address the issue of over fitting our approach to a particular type of API. The threat can be further reduced by evaluating our approach on more subjects.

Threats to internal validity include the correctness of our implementation in extracting usage constraints and labelling a statement as a constraint statement. To reduce the threat, we manually inspected all the constraints inferred against the API method descriptions in our evaluation. Furthermore, we ensured that the results were individually verified and agreed upon by two authors.

If any in-memory buffering is being done by the application (for example, by a `BufferedOutputStream` object), those buffers must be flushed into the `FileDescriptor` (for example, by invoking `OutputStream.flush`) before that data will be affected by `sync`.

If `markposMM` is `-1MM` (no mark has been set or the mark has been invalidated), an `IOExceptionMM` is thrown.

VII. DISCUSSION AND FUTURE WORK

Our approach serves as a way to formalize the description of constraints in the natural language texts of REST API documents (targeted towards generating code contracts), thus facilitating existing tools to process these specifications. We next discuss the benefits of our approach in other areas of software engineering, followed by a description of the limitations of the current implementation and our approach.

A. Limitations:

Leveraging Error Descriptions

`BucketAlreadyExists`: “The requested bucket name is not available. The bucket namespace is shared by all users of the system. Please select a different name and try again.”

Semantic Flow

VIII. CONCLUSION

Although highly desirable, most API’s do not have formal specifications. In contrast documentation of methods contain detailed specifications of the usage in natural language text. However, formal analysis tools are not designed to work on specifications in natural languages. Manually writing formal specifications based on natural language text in API documents is prohibitively time consuming and error prone. To address this issue, we proposed a novel approach ICON to infer temporal constraints from natural language text of API documents. We applied ICON to infer temporal constraints from commonly used package `java.io` from JDK API and Amazon S3 REST API. Our evaluation results show that ICON achieves an average of PP% precision and

RR% recall in inferring TT temporal constraints from more than ZZZZ sentences of API documents.

REFERENCES

- [1] Whyper. <https://sites.google.com/site/whypermission/>.
- [2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.
- [3] B. K. Boguraev. Towards finite-state analysis of lexical cohesion. In *Proc. FSMNLP*, 2000.
- [4] R. P. Buse and W. Weimer. Synthesizing API usage examples. In *Proc. 34th ICSE*, pages 782–792, 2012.
- [5] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proc. 17th ISSA*, pages 273–282, 2008.
- [6] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proc. 30th ICSE*, pages 281–290, 2008.
- [7] M. C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *Proc. LREC*, 2006.
- [8] M. C. de Marneffe and C. D. Manning. The stanford typed dependencies representation. In *Workshop COLING*, 2008.
- [9] U. Dekel and J. D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.
- [10] J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proc. 43rd ACL*, 2005.
- [11] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *Proc. 10th FME*, pages 500–517, 2001.
- [12] V. Gervasi and D. Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transactions Software Engineering Methodologies*, 14:277–330, 2005.
- [13] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *Proc. 31st ICSE*, pages 430–440, 2009.
- [14] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering*, 33:526–543, 2007.
- [15] J. Henkel, C. Reichenbach, and A. Diwan. Developing and debugging algebraic specifications for java classes. *ACM Trans. Softw. Eng. Methodol.*, 17(3):14:1–14:37, 2008.
- [16] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proc. 41st ACL*, pages 423–430, 2003.
- [17] D. Klein and D. Manning, Christopher. Fast exact inference with a factored model for natural language parsing. In *Proc. 15th NIPS*, pages 3 – 10, 2003.
- [18] C. Lee, D. Jin, P. Meredith, and G. Rosu. Towards categorizing and formalizing the JDK API. *Technical Report <http://hdl.handle.net/2142/30006>*, Department of Computer Science, University of Illinois at Urbana-Champaign, 2012.
- [19] H. Lee, Y. Peirsman, A. Chang, N. Chambers, M. Surdeanu, and D. Jurafsky. Stanford’s multi-pass sieve coreference resolution system. In *Proc. CoNLL-2011 Shared Task*, 2011.
- [20] G. Little and R. C. Miller. Keyword programming in Java. In *Proc. 22nd ASE*, pages 84–93, 2007.
- [21] Apache Lucene Core. <http://lucene.apache.org/core/>.
- [22] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. ISSA*, pages 232–242, 2002.
- [23] D. G. Novick and K. Ward. Why don’t people read the manual? In *Proc. 24th ACM*, pages 11–18, 2006.
- [24] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: towards automating risk assessment of mobile applications. In *Proc. 22nd USENIX conference on Security*, pages 527–542, 2013.
- [25] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. 34th ICSE*, 2012.
- [26] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proc. 18th ISSA*, pages 93–104, 2009.
- [27] K. Raghunathan, H. Lee, S. Rangarajan, N. Chambers, M. Surdeanu, D. Jurafsky, and C. D. Manning. A multi-pass sieve for coreference resolution. In *Proc. EMNLP*, 2010.
- [28] B. Rubinger and T. Bultan. Contracting the Facebook API. In *4th AV-WEB*, pages 61–72, 2010.
- [29] A. Sinha, A. M. Paradkar, P. Kumanan, and B. Boguraev. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *Proc. DSN*, pages 327–336, 2009.
- [30] A. Sinha, S. M. Sutton Jr., and A. Paradkar. Text2test: Automated inspection of natural language use cases. In *Proc. ICST*, pages 155–164, 2010.
- [31] J. Slankas and L. Williams. Access control policy extraction from unconstrained natural language text. In *Proc. PASSAT*, 2013.
- [32] The Stanford Natural Language Processing Group, 1999. <http://nlp.stanford.edu/>.
- [33] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icommment: bugs or bad comments?*/. In *21st SOSP*, pages 145–158, 2007.
- [34] L. Tan, Y. Zhou, and Y. Padioleau. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proc. 33rd ICSE*, pages 11–20, 2012.

- [35] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing javadoc comments to detect comment-code inconsistencies. In *Proc. 5th ICST*, April 2012.
- [36] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra. Automating test automation. In *Proc. 34th ICSE*, pages 881–891, 2012.
- [37] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. 22nd ASE*, pages 204–213, 2007.
- [38] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In *Proc. 8th ICFEM*, pages 717–736, 2006.
- [39] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proc. 10th Working Conference on MSR*, pages 319–328, 2013.
- [40] Q. Wu, L. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei. Inferring dependency constraints on parameters for web services. In *Proc. 22nd WWW*, pages 1421–1432, 2013.
- [41] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *Proc. 20th FSE*, pages 12:1–12:11, 2012.
- [42] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending API usage patterns. In *Proc. 23rd ECOOP*, pages 318–343, 2009.
- [43] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. 24th ASE*, pages 307–318, 2009.
- [44] H. Zhou, F. Chen, and H. Yang. Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology. In *Proc. 8th QSIC*, pages 225 –234, 2008.