ICON: Inferring Temporal Constraints from Natural Language API Descriptions

Abstract—**Temporal constraints of an Application Programming Interface (API) are the allowed sequences of method invocations in the API. These constraints govern the secure and robust operation of client software using the API. However, in practice, most API do not have formal temporal constraints. In contrast, these constraints are typically described informally in natural language API documents and therefore are not amenable to existing program analysis tools that require formal constraints. *The goal of this work is to assist developers construct API clients that comply with temporal constraints of the API through the inference and formalization of these constraints found in natural language text in API documents.* Since, API documents are often verbose manually writing formal temporal constraints can be prohibitively time consuming and error prone. To address this issue, we propose ICON, a Machine Learning (ML) and Natural Language Processing (NLP) based approach to identify and infer formal temporal constraints. To evaluate our approach, we apply ICON to infer and formalize temporal constraints from `Amazon S3 REST` API, `PayPal Payment REST` API and commonly used package `java.io` in the JDK API. Our results indicate that ICON is effective in identifying constraint sentences (from over 4000 human-annotated API sentences) with the average precision, recall, and F-score of 65.0%, 72.2%, and 68.4%, respectively. Furthermore, ICON also achieves an accuracy of 70% in inferring 63 formal temporal constraints from these sentences.**

I. Introduction

The increasing reach of the internet has fueled the growth of web Application Programming Interfaces (API). The ProgrammableWeb reports that the web APIs have doubled from Jan 2012 to Oct 2013[1]. These APIs are being combined in interesting ways by application developers creating third-party applications using these APIs to provide value added service. For instance, many online stores leverage Paypal API to assist users with paying for the purchased commodities without actually providing financial information to the online store.

Unlike most traditional API libraries that are in the format of locally stored binary files and are programming language specific, web APIs are designed to leverage the web server and web browser architecture and the implementation is programming language agnostic. However, Web APIs like traditional APIs have rules or constraints governing the proper use of the API that must be followed. One such type of constraints are temporal constraints [2], which are the allowed sequences of invocations of methods from the API. Non-compliance to such constraints can result in faulty applications that are unreliable to use and may even result in financial losses.

[1]http://www.programmableweb.com/api-research

In traditional API, which are usually implemented in typed languages some temporal constraints are enforced by the type system itself. For instance, 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). Thus compliance of such constraints are easily enforced by compilers. In contrast to traditional API libraries, web APIs work on basic parameter data types such as `strings` and `numbers`. Therefore enforcing temporal constraints through type system is ineffective in web APIs. Additionally, temporal constraints are not entirely restricted to type definitions.

Formal analysis tools such as model checker and runtime verifiers can assist in detection of the violations of the temporal constraints in web API clients as defects [16]. However, such tools typically accept formal representation of the temporal constraints for detecting violations. In contrast, temporal constraints are typically described in natural language text of API documents. Such documents are provided to client-code developers through an online access, or are shipped with the API wrapper code. For a method under consideration, an API document may describe both the constraints on the method parameters as well as the temporal constraints in terms of other methods that must be invoked pre/post invoking that method. We observe that a non-trivial portion (roughly 12%) of the sentences (constraint-describing sentences) in `Amazon S3 REST` API documentation describe temporal constraints.

Although natural language API description can be manually converted into formal constraints, manually writing formal constraints based on natural language text in API documents can be prohibitively time consuming and error prone [24], [36]. For instance, Wu et al. [36] report that it took one person ten hours to browse the documentation of one method from a web API, even before the person attempted to formalize the constraints on the method.

To reduce the manual effort, we propose ICON: a Natural Language Processing (NLP) based approach to automatically infer formal temporal constraints. In particular, our approach addresses two major challenges to automatically infer specifications from natural language text in API documents.

First, existing NLP techniques are initially designed towards well-written generic news articles. In contrast, API documents are domain-specific documents and are often not well formed. Furthermore, we observe that generic NLP techniques face problems with lengthy sentences (with a number of lexical tokens). Consider the description sentence “All objects (including all object versions and Delete Markers) in the bucket must be deleted before the bucket itself can be delete.” The length of the sentence may overwhelm an NLP parser, which then inaccurately extracts semantic relations between the words in the sentence. Our ICON approach includes a new technique called "hybrid parsing" to deal with the number of lexical tokens. Hybrid parsing automatically breaks lengthy sentences to smaller tractable sentences based on a function of parts-of-speech tags.

Second, after API documents are made amenable to existing NLP techniques, a method referenced in the natural language text must be identified to infer temporal constraints on the method but such method referencing is often implicit. Consider 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 requires domain dictionaries. Ad-hoc construction of such dictionaries is prohibitively resource intensive and error-prone. To address this challenge, we propose to build domain dictionaries systematically from API documents and generic English dictionaries.

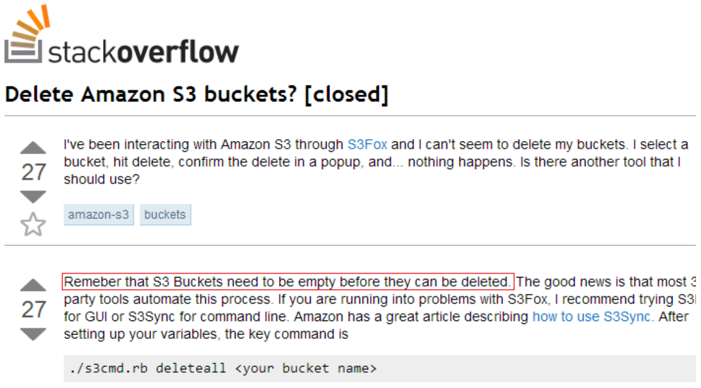
In summary, our ICON approach leverages natural language description of API's to infer temporal constraints of method invocations. As our approach analyzes API documents in natural language, it can be reused independent of the programming language of an API library. Additionally, our approach complements existing mining-based approaches [3], [33], [35], [39] that partially address the problem by mining for common usage patterns among client code reusing the API. This paper makes the following main contributions:

- An NLP-based approach that effectively infers formal temporal constraints of method invocations. To the best of our knowledge, our approach is the first one to apply NLP for inferring temporal constraints from API documents.
- A prototype implementation of our approach based on extending the Stanford Parser [14], 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 website[2], along with the experimental subjects and the results.
- An evaluation of our approach on `Amazon S3 REST` API, `PayPal Payment REST` API, and commonly used package `java.io` from the JDK API.

The rest of the paper is organized as follows. Section II presents a real-world example that motivates our approach. Section III discusses related work. Section IV presents the background on NLP techniques used in our approach. Section V presents our approach. Section VI presents the evaluation of our approach. Section VII presents brief discussion on the limitations and future work. Finally, Section VIII concludes.

II. Motivating Example

We next present a real world example to motivate our approach. Through the example, we demonstrate that developers often ignore the temporal constraints of an API described in the documentation. We suspect the reason for this phenomena is that the documentation is often verbose and the information is distributed across various pages. For instance, the PDF version of the documentation for `Amazon S3 REST` API[3] spans 278 pages. Often developers may not have time (and/or patience) to go through all the documentation and may overlook some temporal constraints of the API, resulting in defective client applications that invoke API methods in sequences prohibited by documentation.


stackoverflow

Delete Amazon S3 buckets? [closed]

27

I've been interacting with Amazon S3 through S3Fox and I can't seem to delete my buckets. I select a bucket, hit delete, confirm the delete in a popup, and... nothing happens. Is there another tool that I should use?

27

Remeber that S3 Buckets need to be empty before they can be deleted. The good news is that most 3 party tools automate this process. If you are running into problems with S3Fox, I recommend trying S3I for GUI or S3Sync for command line. Amazon has a great article describing how to use S3Sync. After setting up your variables, the key command is

```
./s3cmd.rb deleteall <your bucket name>
```

Fig. 1. The Query posted on Stack Overflow forum regrading Amazon S3 REST API

Consider the question asked in *Stack Overflow* [4] as shown in Figure 1. Stack Overflow is an online question and answer forum 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. The inquisitor complains about an issue in the delete bucket functionality of the `S3Fox`. The `S3Fox` developers overlooked the constraints in `Amazon S3 REST` API developer documentation, causing the issue. The API document pertaining to the delete bucket functionality states that before deleting the bucket, the objects in the buckets must be deleted. *"All objects (including all objects versions and Delete Markers) in the bucket must be deleted before the bucket itself can be deleted"*. Although the issue was fixed, one of the forum responses contained a recommendation for the inquisitor to switch to another product. Customer dissatisfaction, such as the one was caused by this issue with the delete bucket functionality, can lead to a loss in revenue.

The presented issue can be easily detected using formal analysis tools. For instance, a specification rule (temporal constraint) can be added to a static checker to verify the presence of a call to delete object functionality before the call to delete bucket functionality. In next section, we briefly discuss the related work pertinent to our approach.

[2]https://sites.google.com/site/temporalspec

[3]http://awsdocs.s3.amazonaws.com/S3/latest/s3-api.pdf

[4]http://stackoverflow.com/

III. Related work

Formal Specification: Contracts are a well-known mechanism for formally specifying functional behavior of the program. Contracts specify the program behavior in terms of conditions that must hold before/after and/or during the execution of a method. A significant amount of work has been done in automated inference of contracts. Existing approaches use program analysis [6], [18], [34] to automatically infer contracts. However, recent studies [10], [22] demonstrate that a combination of developer-written and automatically extracted contracts is the most effective approach for formally specifying the constraints on an API.

Additionally, contracts are typically in the form of assertions on the state (member variables/ properties) of a program. In contrast, temporal constraints specify the ordering of method invocations and therefore are different. Furthermore, since ICON infers temporal constraint from API documents, we believe ICON can work in conjunction with existing approaches to infer a comprehensive set of program specifications.

Furthermore, a different set of approaches exist that infer code-contract-like specifications (such as behavioral model, algebraic specifications, and exception specifications) either dynamically [11]–[13] or statically [4], [10] from source code and binaries. In contrast, ICON infers contracts 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.

NLP in SE: NLP techniques are being increasingly applied in the SE domain. Tan et al. [29] were the first to apply ML and NLP on code comments to detect mismatches between the comments and the implementation. They rely on predefined rule templates targeted towards threading and lock related comments, and then apply ML-based approach to find comments following such rules. The constraints inferred by their approach are the restrictions imposed by the developer on the client code. In comparison, the temporal constraints inferred by ICON are the restriction imposed by the API library being used by the client code.

Zhong et al. [40] also leverage ML along with type information to infer constraints on resources. Specifically, their approach infers resource constraints following template: resource creation methods followed by resource manipulation methods followed by resource release methods. However, temporal constraints may not be limited to such templates. Furthermore, the these approaches rely on specific templates for inferring constraints. In contrast, ICON works independent of such templates for identifying constraints.

Xiao et al. [37] and Slankas et al. [28] 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 satisfactorily on natural language text in use cases, owing to relative well-formed structure of sentences in use case descriptions. In contrast, often the sentences in API documents are not well-formed. Additionally, their approaches do not deal with programming keywords or identifiers, which are often mixed with the method descriptions in API documents.

Pandita et al. [21] proposed an NLP based approach on inferring parameter constraints from method descriptions in the API documents. ICON differs from the their work as follows. ICON addresses the problem of inferring temporal constraint, which is not addressed by the previous approach. ICON significantly extends the infrastructure used by Pandita et al. [21] in following dimensions. First, ICON relies on ML to identify the temporal constraint sentences. The lower frequency of occurrence of temporal constraint sentences in comparison to parameter constraint sentences, make them harder to detect. Second, approach introduces hybrid shallow parsing that relies both on parts-of-speech tags as well as Stanford-typed dependencies to construct intermediate representation, while the previous approach relies only on parts-of-speech tags only. Finally, the ICON approach leverages the concept of semantic graphs constructed from class and method names in API to automatically infer the implicit method references in a sentence.

Augmented Documentation: Improving the documentation related to a software API [9], [30] is another related field of research. Dekel and Herbsleb [9], were the first to create a tool namely eMoose, an Eclipse [5] based 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. [16] 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, ICON 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 ICON.

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], [14], [15] have increased the accuracy of existing NLP techniques to annotate the grammatical structure of a sentence. These advances in NLP have inspired researchers/ practitioners [20], [21], [28], [32], [37] to adapt/apply NLP techniques to solve problems in SE domain.

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 SE specific NLP techniques proposed in related work [20], [21] that are used in this work.

A. Core NLP techniques

Parts Of Speech (POS) tagging [14], [15]. Also known as *'word tagging'*, *'grammatical tagging'* and *'word-sense*

[5]http://www.eclipse.org/

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 demonstrated to be effective 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 effectively classify 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 represents, thus facilitating machine based manipulation of natural language text.

We next present Software Engineering (SE) specific NLP techniques.

B. SE specific NLP techniques

Programming keywords [21]. Accurate annotation of POS tags in a sentence is fundamental to effectiveness of any advanced NLP technique. However POS tagging works satisfactorily on well-written news articles which does not necessary entail that the tagging works satisfactorily on 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 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[6] has eight different definition of word `POST`, and none of them describes `POST` as an HTTP method[7] 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 in turn increasing accuracy of advanced NLP techniques such as chunking and typed dependency annotation.

Lexical Token Reduction [20]. These are a group of generic preprocessing heuristics to further improve the accuracy of core NLP techniques. The accuracy of core NLP techniques is often 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, 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 phenomenon 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 [20]. This technique accepts the syntax-annotated sentences (grammatical and semantic) and builds a First-Order-Logic (FOL) representation of the sentence. Earlier researches have shown the adequacy using FOL for NLP related analysis tasks [21], [26], [27]. In particular, WHYPER [20] demonstrates the effectiveness of this technique, by constructing an intermediate representation generator based on shallow parsing [2] techniques. The shallow parser itself is implemented as a sequence of cascading finite state machines based on the function of Stanford-typed dependencies [7], [8], [14], [15].

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

C. Temporal Specifications

We next describe various classes of temporal constraints using formal temporal logic. Temporal logic is a system of rules and symbols that are used to express and reason about prepositions pertaining to time. We next present an extension to the temporal logic for API method invocation related rules proposed by Lo et al. [17]

[6]http://oxforddictionaries.com/us/definition/american_english/post?q=POST

[7]In 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"*

Method Invocation: We adapt the definition of events proposed by Lo et al. [17] as method invocation. They define event as *"any concrete or abstract representation of a program state"*. We propose the notion of method invocation analogous to the events, where each method invocation is represented as a tuple which is an ordered set of strings. A method invocation is captured as the Tuple `<T> <str1, str2, str3... >`, where `str1, str2, str3`... are string values, that belong to set of all strings. The method invocation tuple `<T>` captures information about the return value, enclosing type, name, and the parameter types of the method. Therefore a tuple `<T> <String, Bar, foo, Integer, Integer>` represents a method `foo` enclosed by the type `Bar`, that accepts two parameters of type `Integer` and a return value of type `String`. For better readability mnemonic representation of the preceding tuple is `String<-Bar.foo(Integer, Integer);`

Method Invocation Predicates: We define method predicate as a predicate E over method invocations. We reuse the notations proposed by Lo et al. [17] $m \vdash \xi$ to denote that method invocation m satisfies method predicate ξ. In context of ICON, method predicates combines a method name and the enclosing type with potential constraints on the return types and parameter types. The predicates are modeled after *equality constraints predicates*, and are represented as `$i=sym`, where `i` is a non-negative integer and `sym` is the primitive string value and defined as: a method invocation m satisfies a predicates `$i=sym` iff `m[i]=sym`. The simplest form of these predicates is of type `$2 = sym` which refer to method invocations of name `sym`.

Temporal Operators: Temporal constraints [8] are constructed by combining method invocation predicates using temporal operators. Lo et al. [17], proposed the following temporal operators:

1) *Forward Eventual Operator* ($\xi_1 \overset{*}{\rightarrow} \xi_2$): occurrence of ξ_1 must be eventually followed by occurrence of ξ_2
2) *Backward Eventual Operator:* ($\xi_1 \overset{*}{\leftarrow} \xi_2$): occurrence of ξ_1 must be preceded by occurrence of ξ_2
3) *Forward Alternation Operator* ($\xi_1 \overset{a}{\rightarrow} \xi_2$): occurrence of ξ_1 must be eventually followed by occurrence of ξ_2 and ξ_1 cannot occur in between
4) *Backward Alternation Operator* ($\xi_1 \overset{a}{\leftarrow} \xi_2$): occurrence of ξ_1 must be preceded by occurrence of ξ_2 and ξ_1 cannot occur in between

Quantification: Quantification is used to introduce constraints involving parameter and return types of a method invocation. Formal representation of temporal constraints are obtained by varying: 1) the temporal operators allowed, b) the method-invocation predicates allowed, and 3) the degree of quantification allowed. For instance, consider the constraint any method in an enclosing type `Obj` must not be invoked after method `foo` in the `Obj`: ∀ `m <m in Obj> !=<foo in Obj>`.

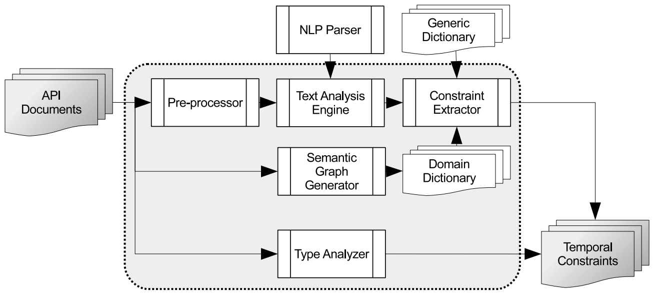


Fig. 2. Overview of ICON approach

V. ICON Overview

We next present our approach for inferring temporal constraints from the method descriptions in API Documents. Figure 2 gives an overview of ICON approach. ICON consists of five major components: a preprocessor, a text-analysis engine, a semantic graph generator, constraint extractor, and a type analyzer. We next describe each component in detail.

A. Preprocessor

The preprocessor accepts the API documents and extracts method descriptions from it. In particular, the preprocessor extracts the following fields from the method descriptions: 1) *Summary of the API method*, 2) *Summary and type information of parameters of the API method*, 3) *Summary and type information of return values of the method*, and 4) *Summary and type information of exceptions thrown (or errors returned) by the methods*.

This step is required to extract the desired descriptive text from the API documents. Different API documents may have different styles of presenting information to developers. This difference in style may also include the difference in the level of detail presented to developers. ICON thus relies on only basic fields that are generally available for API methods across different presentation styles.

After extracting desired information, the natural language text is further preprocessed to be analyzed by subsequent components. The preprocessing steps are required to increase the accuracy of core NLP techniques (described in Section IV-A) that are used in the subsequent phases of ICON approach. The preprocessor first employs the heuristics listed under lexical token reduction, as introduced in Section IV-B.

B. NLP Parser

The NLP parser accepts the pre-processed documents and annotates every sentence in each document using core NLP techniques described in Section IV-A. In particular, each sentence is annotated with: *1)POS tags, 2)named-entity annotations, and 3)Stanford-typed dependencies.* For more details on these techniques and their application, please refer to [7], [8], [20], [21], [32].

Next we use an example to illustrate the annotations added by the NLP Parser. Consider the sentence from the example

[8]Temporal formulae

```
-> deleted-VBN (root)
  -> objects-NNS (nsubjpass)
    -> All-DT (det)
    -> including-VBG (dep)
      -> object versions-NNS (pobj)
        -> all-DT (det)
        -> Delete Markers-NNS (conj_and)
      -> Delete Markers-NNS (pobj)
    -> bucket-NN (prep_in)
      -> the-DT (det)
  -> must-MD (aux)
  -> be-VB (auxpass)
  -> bucket-NN (prep_before)
    -> the-DT (det)
    -> deleted-VBN (rcmod)
      -> itself-PRP (nsubjpass)
      -> can-MD (aux)
      -> be-VB (auxpass)
```

Fig. 3. Sentence annotated with Stanford dependencies

section ***'All objects (including all object versions and Delete Markers) in the bucket must be deleted before the bucket itself can be deleted."***. Figure 3 shows the sentence annotated by NLP parser. Each word (occurs first) is followed by the Part-Of-Speech (POS) tag of the word (in green), which is further followed by the name of Stanford dependency connecting the actual word of the sentence to its predecessor.

C. Candidate Identifier

This component accepts the annotated sentence from the previous component, then using a trained model classifies whether a sentence describes a temporal constraint or not. We next describe the model construction.

The goal of model construction is to use a small set of manually classified temporal constraint sentences of a representative API to construct a model to automatically classify the unlabeled sentences. Since ICON seeks to achieve a binary classification of the API sentences, we chose to train our model using naïve Bayes classifier which is the simplest probabilistic classification methods and is shown to be comparative other advanced classification methods given appropriate prepossessing [23]. The model can be trained offline or else by the users of the approach if better accuracy is required for specific data.

Feature selection is an important factor for the accuracy any classification method. Particularly, for naïve Bayes classifier which assumes each feature independently contributes towards the final probability of classification. In the simplest form, each word occurring in a sentence is considered as a feature. However, such an approach may lead to overly specific models, and therefore ICON approach employs preprocessing for generic features. We next describe the features we chose for training our model along with the rationale for selecting such features.

1) **Length of a Sentence**: The feature is the total number of words in the sentence. The rationale is that shorter sentences (containing fewer words) are unlikely candidates for temporal constraint sentences.
2) **Sentence Type**: The context of sentence, whether the sentence appears as method summary, parameter summary, return value summary or exception/error summary as captured by pre-processor phase. The rationale is that a majority of the temporal constraint sentences are either summary or exception sentences.
3) **Lemmatization**: The feature is the base form of a word. In linguistic lemmatization is the reduction of operational form of a word to its base form. For instance "invoking", "invokes", and "invoked" are all reduced to invoke. The rationale for reducing the words to base form is to reduce the size of the feature set that otherwise considers every word as an independent feature.
4) **Stopword Reduction**: In linguistics stop words are the frequently occurring words that can easily be ignored and are often considered noise, such as "the" , "of", "to" etc... The rationale for filtering the stop words is to reduce the size of feature set that otherwise considers such word as an independent feature but are often noise. Stop word list is further augmented by adding to them the words that occur exactly once in the training corpus to avoid over-fitting of the model to the training corpus.
5) **Stanford Dependencies**: We add to the feature set by identifying the presence of specific Stanford typed dependencies pertaining to the temporal aspects of the sentence semantics. In particular we identify the presence of "*advcl*", "*aux*", "*auxpass*", "*vmod*", and "*tmod*". For instance, the annotated sentence in Figure 3 contains both "*aux*" and "*auxpass*" dependencies. These are both added to the feature set.
6) **POS Tags**: We filter words whose part of speech tags are "*Noun*", "*Determiner*", "*Adjective*", "*Cardinal Number*", "*Foreign Word*", "*Brackets*", "*Coordinating Conjunction*", and "*Personal Pronouns*". The rationale for filtering based on POS tags is to remove the words that are unlikely to be specific to temporal constraints. For instance, presence or absence of determiners is unlikely to affect the outcome of classifier. We further, annotate the POS of word to further distinguish between the words used in different context.

We use this feature set to train a classier to perform binary classification of temporal constraint sentence vs other sentences. The rationale of using ML based approach as opposed to a rule based approach is because: 1) rule-writing requires domain expertise, 2) rules-writing tends to quickly become ad hoc thus requires greater effort to generate generic rules, and 3) ML classifiers have shown to scale well with large volumes of data.

D. Text Analysis Engine

The text analysis engine component accepts the sentences identified as constraint sentences and creates an intermediate representation of each sentence. We define our representation as a tree structure that is essentially a FOL expression. Research literature provides evidence of the adequacy of using FOL for NLP related analysis tasks [20], [21], [26], [27].

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 [2]. In particular, the intermediate-representation technique is implemented as a function of Stanford-typed dependencies [7], [8], [15], to leverage the semantic information encoded in Stanford-typed dependencies.

However, we observed that such implementation is overwhelmed by complex sentences. We improve the accuracy of intermediate-representation generation by proposing a 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 a two phase process as opposed to previous work [20], [21]. 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 which are then accurately annotated by the underlying NLP Parser:

"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 finds it difficult to annotate accurately the Stanford-typed dependencies of the sentence because of presence of different clauses acting on different subject-object pairs. As shown in figure 3 the word including is annotated with Stanford-typed dependencies "dep" that is a catch all dependency. 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[9]. With respect to the previous example the template 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. Figure 4 is the FOL representation of the sentence *"All objects in the bucket must be deleted before the bucket itself can be deleted"*. All the leaf nodes (entities) are represented as the bold words. For readability each node is appended with a number representing the in-order traversal index of the tree.

[9]https://sites.google.com/site/temporalspec

```
before[6]
|->must be deleted[5]
   |->All[1]
   |    |->in[3]
   |        |->objects[2]
   |        |->bucket[4]
   |->can be deleted[8]
       |->bucket[7]
       |->itself[9]
```

Fig. 4. FOL representation of the sentence *"All objects in the bucket must be deleted before the bucket itself can be deleted."*

E. Constraint Extractor

Constraint Extractor is responsible for inferring temporal constraint from the classified constraint sentences. ICON uses the formalism proposed by Lo et al. [17] (introduced in Section IV) for formalizing the constraints. ICON extends the operators by proposing two additional operators:

1) Forward Prohibition Operator ($\xi_1 \overset{-}{\longrightarrow} \xi_2$): occurrence of ξ_1 cannot be followed by occurrence of ξ_2
2) Backward Prohibition Operator ($\xi_1 \overset{-}{\longleftarrow} \xi_2$): occurrence of ξ_1 cannot be preceded by the occurrence of ξ_2

The prohibition operators capture a different class of temporal constraints that cannot be expressed by the formulae constructed by previously proposed operators and their negations. For instance, the *prohibition operators* seem to be negation of *forward eventual operators* and *backward eventual operators*. However, the negation of forward eventual operators is: not an invocation of ξ_1 must be eventually followed by not an invocation of ξ_2. In contrast, *prohibition operators* either negates antecedent or consequent but not both. In summary, we extended the formalism for temporal rules previously proposed by Lo et al. [17] to express temporal constraints.

ICON first identifies ξ_1 as the method whose description constraint sentence is part of. For instance the sentence in Figure 3 is part of Delete Bucket method description in `Amazon S3 REST` API, ξ_1 is instantiated as Delete Bucket method.

ICON next identifies ξ_2. Since, references to ξ_2 may not always be implicit we leverage the semantic graphs. A semantic graph is a representation of concepts of objects and the methods applicable on those objects. Figure 5 shows a graph for `Object` resource in `Amazon S3 REST` API. The phrases in rounded rectangle are the actions applicable on `Object` resource. Section V-F further describes how these graphs are generated. ICON uses the Algorithm 1 to identify ξ_2.

The algorithm systematically explores the FOL representation of the candidate sentence to identify ξ_2. First, the algorithm attempts to locate the occurrence of object name or its synonym in the leaf nodes of the FOL representation of the sentence (Line 3). The method findLeafContain-

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.

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: $FOL\_rep\ r' = rep.findLeafContaining(r\_nam\_list)$
 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 $action$
```

ing(r_name_list) explores the FOL representation to find a leaf node that contains either the object name or one of its synonyms. Once a leaf node is found, we systematically traverse the tree from the leaf node to the root, matching all parent predicates as well as immediate child predicates [Lines 5-16]. Algorithm matches each of the traversed predicate with the actions associated with the object defined in semantic graph. ICON further employs WordNet and Lemmatisation to deal with synonyms to find appropriate matches. If a match is found, then the matching action is returned as ξ_2. ICON does not consider self references, that is if $\xi_2 = \xi_1$, the identified method reference is discarded. In case of multiple matching actions ICON considers only the first match. For instance the sentence in Figure 3 algorithm identifies Delete Object method as ξ_2

ICON next identifies the direction (forward or backward) of the relationship by examining the tense of ξ_2 reference in the sentence. Past tense is considered as backward and other tenses are considered as forward. For instance, the sentence in Figure 3 since “deleted” is in past tense and therefore operator is backward and the constraint is Delete Object $\overset{*}{\leftarrow}$ Delete Method.

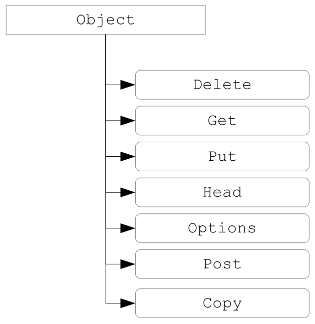


Fig. 5. Semantic Graph for the `Object` related operations in Amazon S3 REST API

The negation is identified by presence of negation verbs such as “*no*”, “*not*”, “*can’t*” ... etc. Another rule for negation operator is if the sentence is in exception/error description.

F. Semantic-Graph Generator

A key way of identifying reference to a method in the API by ICON is the employment of a semantic graph of an API. We propose to initially infer such graphs from API documents. Ad hoc creation of semantic graph is prohibitively time consuming and may be error prone. We thus employ a systematic methodology (proposed previously by Pandita et al. [20]) to infer semantic graphs from API documents.

We first consider the name of the class for the API document in question. 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. In case of REST API we first identified the resources and then listed REST actions on those resources as applicable actions. Figure 5 shows the graph for `Object` resource in

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:                 $List\ preList\ =\ idx.query(paramType)$
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 $seq\_Graph$
```

REST API. The phrases in rounded rectangle are the REST actions applicable on `Object` resource in `Amazon S3 REST` API.

G. Type Analysis

Some temporal constraints are enforced by the type system in typed Languages. 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). To extend the temporal constraints inferred by the analyzing the natural language text, this component infers additional constraints that are encoded in the type system. Algorithm 2 lists the steps followed to infer type based temporal constraints.

The algorithm accepts the list of methods as an input produces a graph with the nodes representing methods in an API and the directed edges representing temporal constraints. First, an index is created based on the return types of the method (Line 2). Second, all methods in an API are added to an unconnected graph (Line 3-4). Then, for every public method in the input list, the algorithm checks the types of the input parameters and constructs and directed edge from all the methods whose return value have the same type to the method in question (Line 14- 20). The algorithm does not take into consideration the basic parameter types such as `integer`, `string` (Line 15). Additionally, an edge is created from the constructors of a class to the non static members methods of a class (Line 8 -13). The resultant graph is then returned by the algorithm.

The temporal constraints based on the type information can be extracted by querying the graph. The incoming edges to a node denoting a method represents the set of pre-requisite methods. The temporal constraint being, at least one of the pre-requisite methods must be invoked before invoking the method in question.

VI. Evaluation

We next present the evaluation we conducted to assess the effectiveness of ICON. In our evaluation, we address three main research questions:

- **RQ1**: What are the precision and recall of ICON in identifying temporal constraints from sentences written in natural language? Answer to this question quantifies the effectiveness of ICON in identifying constraint sentences.
- **RQ2**: What is the accuracy of ICON in inferring temporal constraints from constraint sentences in the API documents? Answer to this question quantifies the effectiveness of ICON in inferring temporal constraints from constraint sentences.
- **RQ3**: What is the degree of the overlap between the temporal constraints inferred from natural language text in comparison to the typed-enforced temporal constraints?

A. Subjects

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

- `Amazon S3 REST` API provides a REST based web services interface 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, and inexpensive infrastructure. `Amazon S3` is reported to store more than 2 trillion objects as of April 2013 and gets over 1.1 million requests per second at peak time [1].
- `PayPal Payment REST` API provides a REST based web service interface to facilitate online payments and money transfer. `PayPal` reports to have handled \$56.6 billion(USD) worth of transactions (total value of transactions) in just the third quarter of 2014.
- `java.io` : is one of a popular packages in `Java` programming language. The package provides APIs for system input and output through data streams, serialization and the file system, which are one of the fundamental functionalities provided by a programming language.

We chose `Amazon S3`, (PayPal) and `java.io` APIs as our subjects because they are popular and contain decent documentation.

B. Evaluation Setup

We first manually annotated the sentences in the API documents of the two APIs. Two authors manually labeled each sentence in the API documentation as sentence containing temporal constraints or not. We used `cohen kappa` [5] score to statistically measure the inter-rater agreement. The `cohen kappa` score of the two authors was .66 (on a scale of 0 to 1), which denotes a statically significant agreement [5]. After the authors classified all the sentences, they discussed with each other to reach a consensus on the sentences they classified differently. We use these classified sentences as the golden set for calculating precision and recall.

To answer RQ1, we measure the number of true positives (TP), false positives (FP), true negative (TN), and false negatives (FN) in identifying the constraint sentences by ICON. We define constraint sentence as a sentence describing a temporal constraints. We define the TP, FP, TN, and FN of ICON as follows:

1) TP: A sentence correctly identified by ICON as constraint sentence.
2) FP: A sentence incorrectly identified by ICON as constraint sentence.
3) TN: A sentence correctly identified by ICON as not a constraint sentence.
4) FN: A sentence incorrectly identified by ICON as not a constraint sentence.

In statistical classification [19], `precision` is defined as a ratio of number of true positives to the total number of items reported to be true, `recall` is defined as a ratio of number of true positives to the total number of items that are true. `F-Score` is defined as the weighted harmonic mean of `precision` and `recall`. Higher value of `precision`, `recall`, and `F-Score` are indicative of higher quality of the constraint statements inferred using ICON. based on the calculation of TP, FP, TN, and FN of ICON defined previously we computed the `precision`, `recall`, and `F-Score` of ICON as follows:

$$Precision = \frac{TP}{TP+FP}$$
$$Recall = \frac{TP}{TP+FN}$$
$$F-Score = \frac{2 X Precison X Recall}{Precision+Recall}$$

To answer RQ2, we manually verified the temporal constraints inferred from constraint sentences by ICON. However, we excluded the type-enforced temporal constraints inferred using Algorithm 2, described in Section V. We excluded the type-enforced constraints because they are correct by construction and are by default enforced by modern IDE's such as eclipse. We then measure *accuracy* of ICON as the ratio of the total number of temporal constraints that are correctly inferred by ICON to the total number of constraint sentences. Two authors independently verified the correctness of the temporal constraints inferred by ICON. We define the `accuracy` of ICON as the ratio of constraint sentences with correctly inferred temporal constraints to the total number of constraint sentences. Higher value of `accuracy` is indicative of effectiveness of ICON in inferring temporal constraints from constraint sentences.

To answer RQ3, we counted the overlap in the temporal constraints inferred by ICON from the natural language text in API documents to the type-enforced temporal constraints inferred using Algorithm 2, described in Section V.

C. Results

We next present our evaluation results.

1) RQ1: Effectiveness in Identifying Constraint Sentences: In this section, we quantify the effectiveness of ICON in identifying constraint sentences by answering RQ1. Table II shows the effectiveness of ICON in identifying constraint sentences. Column "API" lists the names of the subject API. Columns "Mtds" and "Sen" lists the number of methods and sentences in each subject API's. Column "Sen$_C$" list the number of manually identified constraint sentences. Column "Sen$_{ICON}$" lists the number of sentences identified by ICON as constraint sentences. Columns "TP", "FP", "TN", and "FN" represent the number of `true positives`, `false positives`, `true negatives`, and `false negatives`, respectively. Columns "P(%)", "R(%)", and "F$_S$(%)" list percentage values of `precision`, `recall`, and `F-score` respectively. Our results show that, out of 3,909 sentences, ICON effectively identifies constraint sentences with the average `precision`, `recall`, and `F-score` of 65.0%, 72.2%, and 68.4%, respectively.

We next present an example to illustrate how ICON incorrectly identifies a sentence as a constraint sentence (producing false positives). For instance, consider the sentence "*This is done by flushing the stream and then closing the underlying output stream.*" from `close` method description from `PrintStream` class. ICON incorrectly identifies the action "flush" being performed before the action "close". However ICON fails to make the distinction that it happens internally (enforced in the body) in the method. ICON, thus incorrectly identifies the sentence as a constraint sentence.

Another major source of FPs is the incorrect parsing of sentences by the underlying NLP infrastructure and/or inadequacy of generic dictionaries for synonym analysis. For instance, consider the sentence "*If this stream has an associated channel then the channel is closed as well.*" from the `close` method description from `FileOutputStream`. The sentence describes an effect that happens as a result of calling the `close` method and does not describe any temporal constraint. However, ICON annotates the sentence as a constraint sentence because underlying Wordnet dictionaries matches the word "has" as a synonym of "get". This incorrect matching in turn causes ICON to incorrectly annotate the sentence as constraint sentence because "has" is matched against `(get)` method in `FileOutputStream`. We observed 8 instances of previously described example in our results.

If we manually fixed the Wordnet dictionaries to not match "has" and "get" as synonyms, our precision is further increased to 70.8% effectively increasing the F-Score of ICON to 71.2%. Although an easy fix, we refrained from including such modifications for reporting the results to stay true to our approach. In the future, we plan to investigate techniques to construct better domain dictionaries for software API.

We next present an example to illustrate how ICON fails identify a constraint sentence (producing false negative). False negatives are undesirable in the context of our problem domain because they can mislead the users of ICON into believing that no other temporal constraint exists in the API documents. Furthermore, an overwhelming number of false negatives works against the practicality of ICON. For instance, consider the sentence "*This implementation of the PUT operation creates a copy of an object that is already stored in Amazon S3.*" from `PUT Object-Copy` method description in `Amazon S3`

TABLE II
EVALUATION RESULTS

API	Mtds	Sen	Sen$_C$	Sen$_{ICON}$	TP	FP	FN	P(%)	R(%)	F$_S$(%)	Spec$_{ICON}$	Acc(%)
java.io	662	2417	78	88	57	31	21	64.8	73.1	68.8	56	71.8
`Amazon S3 REST` API	51	1492	12	12	8	4	4	66.7	66.7	66.7	7	58.3
`PayPal Payment REST` API	33	151	20									
Total	746	4060	90	100	65	35	25	65.0*	72.2*	68.4*	63	70.0*

* Column average; Mtds: Total no. of Methods; Sen: Total no. of Sentences; Sen$_C$: Total no. of constraint Sentences; Sen$_{ICON}$: Total no. of constraint Sentences identified by ICON; TP: Total no. of True Positives; FP: Total no. of False Positives; FN: Total no. of False Negatives; P: Precision; R: Recall; F$_S$: F-Score; Acc: Accuracy Spec$_{ICON}$: Total no. of temporal constraint correctly identified by ICON;

`REST` API. The sentence describes the constraint that the object must already be stored (invocation of `PUT Object`) before calling the current method. However, ICONcannot make the connection owing to the limitation of the semantic graphs that do not list "already stored" as a "valid operation" on object. In the future, we plan to investigate techniques to further improve knowledge graphs to infer such implicit constraints.

Another major source of false negatives (similar to reasons for false positives) is the incorrect parsing of sentences by the underlying NLP infrastructure. For instance, consider the sentence "*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.*" The sentence describes that the `OutputStream.flush()` must be invoked before invoking the current method if in-memory buffering is performed. However, the length and complexity in terms of number of clauses causes the underlying Stanford parser to inaccurately annotate the dependencies, which eventually results into incorrect classification.

Overall, a significant number of false positives and false negatives will be reduced as the current NLP research advances the underlying NLP infrastructure. Furthermore, use of domain specific dictionaries as opposed to generic dictionaries sed in current prototype implementation will further improve the precision and recall of ICON.

2) RQ2: Accuracy in Inferring Temporal Constraints: In this section, we evaluate the effectiveness of ICON in inferring temporal constraints from the identified constraint sentences from API documents.Table II shows the effectiveness of ICON in inferring temporal constraints from the identified constraint sentences. Column "API" lists the names of the subject API. Columns "Mtds" and "Sen" list the number of methods and sentences in each subject API's. Column "Sen$_C$" lists the number of manually identified constraint sentences. Column "Spec$_{ICON}$" lists the number of sentences with correctly inferred temporal constraints by ICON. Column "Acc(%)" list percentage values of accuracy. Our results show that, out of 90 manually identified constraint sentences, ICON correctly infers temporal constraints with the average accuracy of 70.0%.

We next present an example to illustrate how ICON incorrectly infers temporal constraints from a constraint sentence. Consider the sentence "*if the stream does not support seek, or if this input stream has been closed by invoking its close method, or an I/O error occurs.*" from `skip` method of `java.io.FilterInputStream` class. Although ICON correctly infers that method `close` cannot be called before current method, ICON incorrectly associates the phrase "support seek" with method `markSupported` in the class. The faulty association happens due to incorrect parsing of the sentence by the underlying NLP infrastructure. Such issues will be alleviated as the underlying NLP infrastructure improves.

Another, major cause of failure for ICON in inferring temporal constraints from sentences is the failure to identify the sentence as a constraint sentences at the first place (false negatives). Overall, accuracy of ICON can be significantly improved by lowering the false negative rate in identifying the constraint sentences.

3) RQ3: Comparison to Typed-Enforced Constraints: In this section, we compared the temporal constraints inferred from the natural language API descriptions to those enforced by the type-system (referred to as type-enforced constraint). The constraints that are enforced by the type-system can be enforced by IDEs. Hence, for such types of constraints, we do not require sophisticated techniques like ICON. For `java.io`, we define a type-enforced constraint as a constraint that mandates a method M accepting input parameter I of type T to be invoked after (at least one) a method M' whose return value is of type T. Since there are no types in REST APIs, for `Amazon S3`, we consider a constraint as a type-enforced constraint if the constraint is implicit in the `CRUD` semantic followed by REST operations. `CRUD` stands for resource manipulation semantic sequence create, retrieve, update, and delete. In particular, we consider a constraint as a type-enforced constraint, if the constraint mandates a DELETE, GET, or PUT operation on a resource to be invoked after a POST operation on the same resource.

To address this question, we manually inspect each of the constraints reported by ICON and classify it as a type-enforced constraint or a non type-enforced constraint. We observed that none of the constraints inferred by our ICON from natural language text were classified as a type-enforced constraint. Hence, the constraints detected by ICON are not trivial enough to be enforced by a type system.

D. Summary

In summary, we demonstrate that ICON effectively identifies constraint sentences (from over 3900 API sentences) with the average precision, recall, and F-score of 65.0%, 72.2%, and 68.4% respectively. Furthermore, we also show that ICON infers temporal constraints from the constraint sentences an average accuracy of 70%. Furthermore, also provide discussion that a false positives rate and false negatives rate can be further improved by improving the underlying NLP infrastructure. Finally, we provide a comparison of the temporal constraints inferred from natural language description against the temporal constraints enforced by a type system.

E. Threats to Validity

Threats to external validity primarily include the degree to which the subject documents used in our evaluation are representative of true practice. To minimize the threat, we used API documents of two different API's: JDK `java.io` and `Amazon S3 REST` API developer documentation. On one hand, Java is a widely used programming language and `java.io` and is one of the main packages. In contrast, `Amazon S3 REST` API developer documentation provides HTTP based access to online storage allowing developers the freedom to write clients applications in any programming language. Furthermore, the difference in the functionalities provided by the two API's 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 API's.

Threats to internal validity include the correctness of our prototype implementation in extracting temporal constraints and labeling 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, using the `cohen kappa` [5] score to statistically measure the inter-rater agreement.

VII. Limitations and Future work

Our approach serves as a way to formalize the description of constraints in the natural language texts of API documents, thus facilitating existing tools to process these specifications. We next discuss some of the limitations of our approach.

Validation of Method Descriptions. API documents can sometimes be misleading [25], [31], thus causes developers to write faulty client code. In future work, we plan to extend our approach to find documentation-implementation inconsistencies.

Inferring Implicit Constraints. The approach presented in this work only infers temporal constraints explicitly described in the method descriptions. However, there are instances where the constraints are implicit. For instance, consider the method description for `markSupported` method in `BufferInputSttream` class in Java, which states "*Test if this input stream supports* `mark`". For a developer it is straightforward to understand that the method `markSupported` must be invoked before the method `mark`. Our approach is unable to infer such implicit temporal constraints. In future work, we plan to investigate techniques to infer these implicit temporal constraints.

Extending Generic Dictionaries. The use of generic dictionaries for software engineering related text is sometimes inadequate. For instance, Wordnet matches "has" as a synonym for the word "get". Although, valid for generic English, such instances cause our approach to incorrectly distinguish a constraint sentence from a regular sentence, or vice versa. In future work, we plan to investigate techniques to extend generic dictionaries for software engineering related text. In particular, Yang and Tan [38] recently proposed a technique for inferring semantically similar words from software context to facilitate code search. We plan to explore such techniques and evaluate the overall effectiveness of our approach after augmenting it with such techniques.

VIII. Conclusion

Despite being highly desirable, most APIs are not equipped with formal temporal constraints. In contrast, documentation of API methods contains detailed specifications of temporal constraints in natural language text. Manually writing formal specifications based on natural language text in API documents is prohibitively time consuming and error prone. To address this issue, we have proposed a novel approach called ICON to infer temporal constraints from natural language text of API documents. We applied ICON to infer temporal constraints from `PayPal Payment REST` API, `Amazon S3 REST` API, and commonly used package `java.io` in the JDK API. Our evaluation results show that ICON effectively identifies sentences describing temporal constraints with an average 65% precision and 72% recall, from more than 4000 sentences in subject API documents. Furthermore, ICON also achieved an accuracy of 70% in inferring 63 formal temporal constraints from these sentences.

References

[1] Amazon S3 - Two Trillion Objects, 1.1 Million Requests / Second. http://aws.typepad.com/aws/2013/04/amazon-s3-two-trillion-objects-11-million-requests-second.html.

[2] B. K. Boguraev. Towards finite-state analysis of lexical cohesion. In *Proc. FSMNLP*, 2000.

[3] R. P. Buse and W. Weimer. Synthesizing API usage examples. In *Proc. 34th ICSE*, pages 782–792, 2012.

[4] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proc. 17th ISSTA*, pages 273–282, 2008.

[5] J. Carletta. Assessing agreement on classification tasks: the kappa statistic. *Computational linguistics*, 22(2):249–254, 1996.

[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] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. 10th FME*, pages 500–517, 2001.

[11] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *Proc. 31st ICSE*, pages 430–440, 2009.

[12] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering*, 33:526–543, 2007.

[13] 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.

[14] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proc. 41st ACL*, pages 423–430, 2003.

[15] D. Klein and C. D. Manning. Fast exact inference with a factored model for natural language parsing. In *Proc. 15th NIPS*, pages 3 – 10, 2003.

[16] 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.

[17] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Mining quantified temporal rules: formalism, algorithms, and evaluation. In *Proc. 16th WCRE*, pages 62–71. IEEE, 2009.

[18] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. ISSTA*, pages 232–242, 2002.

[19] D. Olson. *Advanced data mining techniques*. Springer Verlag, 2008.

[20] 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.

[21] 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.

[22] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proc. 18th ISSTA*, pages 93–104, 2009.

[23] J. D. Rennie, L. Shih, J. Teevan, D. R. Karger, et al. Tackling the poor assumptions of naive bayes text classifiers. In *Proc. 20th ICML*, pages 616–623, 2003.

[24] B. Rubinger and T. Bultan. Contracting the Facebook API. In *4th AV-WEB*, pages 61–72, 2010.

[25] C. Rubino-González and B. Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. In *Proc. 9th PASTE*, pages 73–80, 2010.

[26] 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.

[27] A. Sinha, S. M. SuttonJr., and A. Paradkar. Text2test: Automated inspection of natural language use cases. In *Proc. ICST*, pages 155–164, 2010.

[28] J. Slankas and L. Williams. Access control policy extraction from unconstrained natural language text. In *Proc. PASSAT*, 2013.

[29] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icomment: bugs or bad comments?*/. In *21st SOSP*, pages 145–158, 2007.

[30] 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.

[31] 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.

[32] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra. Automating test automation. In *Proc. 34th ICSE*, pages 881–891, 2012.

[33] 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.

[34] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In *Proc. 8th ICFEM*, pages 717–736, 2006.

[35] 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.

[36] 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.

[37] 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.

[38] J. Yang and L. Tan. SWordNet: Inferring semantically related words from software context. *Empirical Software Engineering*, 2013.

[39] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending API usage patterns. In *Pro. 23rd ECOOP*, pages 318–343, 2009.

[40] 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.