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 APIs do not come with formal temporal constraints. In contrast, these constraints are typically described informally in natural language API documents, and therefore are not amenable to existing tools for checking formal temporal constraints.** ***The goal of this work is to assist developers to construct API clients that comply with temporal constraints of the API through the inference and formalization of these constraints found in natural language API documents.*** **Since API documents are often verbose, manually identifying and writing formal temporal constraints can be prohibitively time-consuming and error-prone. To address this issue, we propose ICON: an approach based on Machine Learning (ML) and Natural Language Processing (NLP) for identifying and inferring formal temporal constraints. To evaluate our approach, we apply ICON to infer and formalize temporal constraints from the `Amazon S3 REST` API, the `PayPal Payment REST` API and the `java.io` package in the JDK API. Our results indicate that ICON is effective in identifying temporal constraint sentences (from over 4000 human-annotated API sentences) with the average precision, recall, and F-score of 79.0%, 60.0%, and 65.0%, respectively. Furthermore, our evaluation also demonstrates that ICON achieves an accuracy of 70% in inferring and formalizing 77 temporal constraints from these temporal constraint sentences.**

I. Introduction

Application Programming Interfaces (APIs) facilitates software reuse by providing standardized mechanism to access API components. However, an API has some 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 will often result in faulty applications that are unreliable to use.

Formal analysis tools, such as model checkers and runtime verifiers can assist in detecting violations of the temporal constraints in API clients as defects [17]. These tools typically accept formal representation of the temporal constraints for detecting violations. However, 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 code. For a method under consideration, 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.

Although natural language API descriptions can be manually converted to formal constraints, manually identifying and writing formal constraints based on natural language text in API documents can be prohibitively time-consuming and error-prone [25], [38]. For instance, the PDF version of the documentation for `Amazon S3 REST` API[1] spans 278 pages describing 51 methods.

The goal of this work is to assist developers to construct API clients that comply with temporal constraints of the API through the inference and formalization of these constraints found in natural language API documents.

The first step towards formalizing temporal constraints is to identify the sentences describing such constraints. A naive approach to identify the constraint sentences is to perform keyword-based search on API documents. However, the effectiveness of such approach is limited by the quality of the keywords used. Furthermore, sentences involving temporal constraints often the time may not have uniquely identifiable keywords.

For instance, consider the sentences from the `PayPal Payment REST` API: 1) "*Use this call to complete a payment.*" from the `execute payment` method. 2) "*Use this call to refund a completed payment.*" from the `refund sale` method. Sentence 1 is a descriptive statement about the `execute payment` method. In contrast, Sentence 2 indicates the temporal constraint that a payment must be completed before the refund call is initiated. Since these sentences are not significantly different in terms of words, a simple keyword-based search will fail to distinguish between the two. Another problem with keyword-based search is that the method names (and synonyms) are part of the keywords themselves, thus resulting in a large number of keywords to be searched. The large number of keywords further negatively affects accuracy.

Even after a sentence has been identified as a constraint sentence, an approach has to infer the references to the method in the sentences, which often may not be explicit. Consider Sentence 2 in the preceding paragraph. The phrase *"completed payment"* refers to the `execute payment` method in the API.

To address these issues, we propose ICON: an approach based on Machine Learning (ML) and Natural Language Processing (NLP) for identifying and inferring formal temporal constraints. We propose to first employ ML for identifying temporal constraint sentences and then use NLP techniques to infer formal temporal constraints from the identified sentences.

An ML based approach for identifying the temporal constraint sentences addresses the limitations of keyword-based search by automatically learning patterns of temporal con-

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

straint sentences. Furthermore, we propose to use a combination of words, syntax (parts of speech) of a sentence, and semantics (relationship between words) of a sentence as the features for learning patterns. This combination allows ICON to make a finer grained distinction between the example sentences described earlier. Furthermore, to identify the phrases as implicit method-invocation references, we propose to leverage domain dictionaries that are systematically created from API documents and generic English dictionaries.

In summary, the ICON approach leverages natural language description of an API 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], [35], [37], [41] that partially address the problem by mining for common usage patterns among client code reusing the API. Our results indicate that ICON is effective in identifying temporal constraint sentences (from over 4000 human-annotated API sentences) with the average precision, recall, and F-score of 79.0%, 60.0%, and 65.0%, respectively. Furthermore, our evaluation also demonstrates that ICON achieves an accuracy of 70% in inferring 77 formal temporal constraints from these sentences. This paper makes the following main contributions:

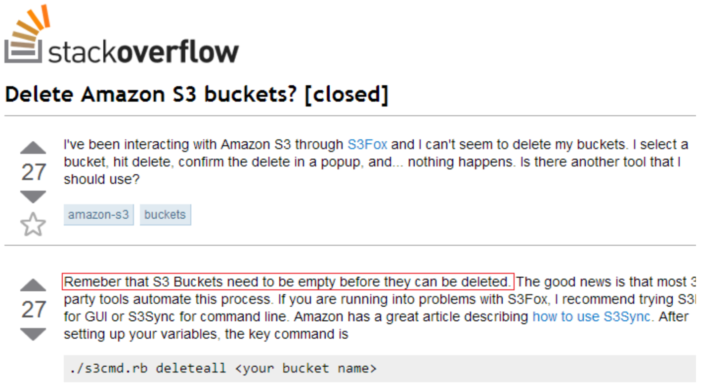
- An ML and NLP-based approach that effectively infers formal temporal constraints of method invocations.
- A prototype implementation of our approach based on extending the Stanford Parser [15], which is a natural language parser to derive the grammatical structure of sentences. An open source implementation 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 the `Amazon S3 REST` API, the `PayPal Payment REST` API, and the 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 behavior is that the documentation is often verbose and the information is distributed across various pages. Often developers may not have time (and/or patience) to go through all the documentation and may overlook some temporal constraints of the

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


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?

amazon-s3 buckets

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 S3 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

API, resulting in defective client applications that invoke API methods in sequences prohibited by documentation.

Consider the question asked in *Stack Overflow* [3] 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 documents, 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 recommended the inquisitor to switch to another product. Customer dissatisfaction caused by such issues 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. We next briefly discuss the related work pertinent to our approach.

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], [19], [36] to automatically infer contracts. However, recent studies [10], [23] demonstrate that a combination of developer-written and automatically extracted

[3]http://stackoverflow.com/questions/27267/delete-amazon-s3-buckets

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, therefore are different. Furthermore, since ICON infers temporal constraints from API documents, we envision ICON to work in conjunction with existing approaches to infer a comprehensive formal specification.

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], [14] or statically [4], [10] from source code and binaries. In contrast, ICON infers contracts from the natural language text in API documents, thus complementing existing approaches when the source code or binaries of the API library is not available.

NLP in Software Engineering (SE):
Research advances [8], [16] in the accuracy of existing NLP techniques have inspired researchers and practitioners [21], [22], [29], [34], [39] to adapt and(/or) apply NLP techniques to solve problems in SE domain. Tan et al. [31] 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. [42] also leverage ML along with type information to infer constraints on resources from API documents. Specifically, their approach infers resource constraints following the template - "*resource creation methods* followed by *resource manipulation methods* followed by *resource release methods*". However, temporal constraints are often not be limited to such template. 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. [39] and Slankas et al. [29] use shallow parsing techniques to infer Access Control Policy (ACP) rules from natural language text in use cases. In contrast, the ICON approach works with API documents. Pandita et al. [22] proposed an NLP-based approach on inferring parameter constraints from method descriptions in the API documents. ICON differs from 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. [22] 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, the ICON 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. 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], [32] is another related field of research. Dekel and Herbsleb [9], were the first to create a tool namely eMoose, an Eclipse [4] 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. [17] 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

We next briefly introduce the techniques used in this work that have been grouped into two 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 [21], [22] that are leveraged in this work.

A. Core NLP techniques

Parts Of Speech (POS) tagging [15], [16]. 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 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 further enhances the syntax of a sentence in addition to 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 SE specific NLP techniques.

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

B. SE specific NLP techniques

Programming keywords [22]. 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.

For instance, consider the word `POST`. The online Oxford dictionary[5] has eight different definition of word `POST`, and none of them describes `POST` as an HTTP method[6] 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 increasing accuracy of advanced NLP techniques such as chunking and typed dependency annotation.

Lexical Token Reduction [21]. These are a collection 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 related work [21], [22] to achieve the desired reduction of lexical tokens:

- **Period Handling**. Besides marking the end of a sentence in generic 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 interleaved with text. This interleaving may result in POS tagger 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.

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

[6]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"*

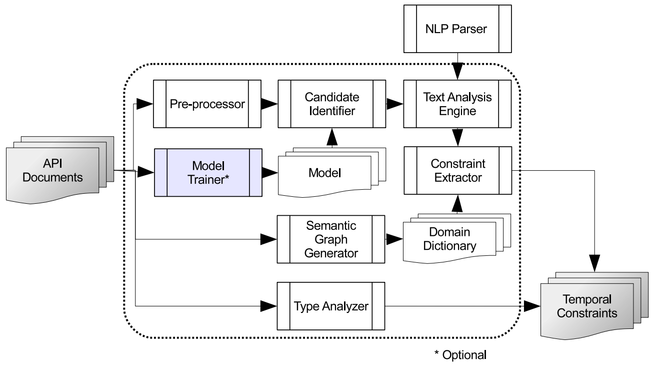


Fig. 2. Overview of the ICON approach

Intermediate-Representation Generation [21]. 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 [22], [27], [28]. Particularly, WHYPER [21] demonstrates the effectiveness of this technique, by constructing an intermediate representation generator from semantically annotated sentences. The component is implemented as a sequence of cascading finite state machines based on the function of annotated Stanford-typed dependencies [7], [8], [15], [16].

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

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 the ICON approach. ICON consists of following six major components: a preprocessor, a candidate identifier, a text-analysis engine, a semantic graph generator, constraint extractor, and a type analyzer. Additionally, there is an optional model trainer component and external NLP parser component.

First, the preprocessor accepts API documents and preprocesses the sentences in the method description. Next, an NLP parser annotates the syntax and semantics of preprocessed sentences. The annotated sentences are accepted by candidate identifier component to classify temporal constraint sentences, using the model trained by the model trainer component. The text-analysis engine further transforms the identified constraint

```
-> 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

sentences into the first-order-logic (FOL) representation. Finally, the constraint extractor leverages the semantic graphs to infer temporal constraints from the FOL representation of a sentence. The type analyzer component infers temporal constraints encoded in the type system of a language by analyzing the API methods parameter and return types. We next describe each component in detail.

A. Preprocessor

The preprocessor accepts the API documents and extracts method descriptions. 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 the details presented to developers. ICON relies only on 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 the 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 [7], [8], [21], [22], [34] described in Section IV-A. In particular, each sentence is annotated with: *1) POS tags, 2) named-entity annotations, and 3) Stanford-typed dependencies.*

Next we use an example to illustrate the annotations added by the NLP Parser. Consider the sentence from the example 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. From an implementation perspective, this component can be implemented using any existing NLP libraries or approaches such as Stanford Parser [30].

C. Candidate Identifier

This component accepts the annotated sentence from the previous component, then using a trained ML 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 classify the unlabeled sentences as being temporal constraints or not. From an implementation perspective, we can use any of the standard off-the-shelf classifiers to train the ML model, since ICON seeks to achieve only a binary classification of the API sentences. For instance, model can be trained using naïve Bayes classifier which is the simplest probabilistic classification methods and is shown to be comparative to other advanced classification methods given appropriate prepossessing [24]. The ML model can be trained offline or else by the users of the approach if better accuracy is desired for domain-specific data.

Feature selection is an important factor for the accuracy any classification method. In the simplest form, each word occurring in a sentence is considered as a feature. However, such an approach may lead to overly specific ML models, and therefore the ICON approach extracts generic features from sentences. We next describe the features we chose for training our ML 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, stopwords are the frequently occurring words that can be ignored and are

often considered noise, such as "the" , "of", "to" etc... The rationale for filtering the stopwords is to reduce the size of feature set that otherwise considers such words (despite being noise) as an independent features. Stopword 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.
7) **Sentence Structure**: The feature is the ordered sequence of chunking tags [15], [16] in a sentence. Chunking seeks to divide a sentence into a constituent set of words (or phrases) that logically belong together (such as a Noun Phrase and Verb Phrase). Thus chunking captures the structure of a sentence. Rationale for selecting ordered sequence of chunking tags is to incorporate structure of a sentence as a feature.

We use the described 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. This intermediate representation is leveraged by subsequent component to infer formal constraints. 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 [21], [22], [27], [28].

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], [16], 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 [21], [22]. 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. A catch all dependency is selected by a parser when no other appropriate dependency can be selected. The ICON approach thus automatically 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 lists the semantic templates used in this phase. Column "Template" describes conditions where the template is applicable and Column "Summary" describes the action taken by ICON when the template is applicable. 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: After complex sentences (whenever applicable) have been broken to simple sentences, this phase generates an intermediate (FOL) representation of the sentences. 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*

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.

```
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."*

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.

E. Constraint Extractor

Constraint Extractor is responsible for inferring temporal constraint from the classified constraint sentences.

Temporal constraints are expressed as temporal formulae involving: 1) *Predicates* ξ representing method calls and 2) *Temporal operators*: backward ($\leftarrow$) & forward ($\rightarrow$) and their negations $\overset{-}{\longleftarrow}$ & $\overset{-}{\longrightarrow}$. We define following four constraints:

1) *Forward Operator* ($\xi_1 \rightarrow \xi_2$): method call ξ_1 must be succeeded by method call ξ_2.
2) *Backward Operator* ($\xi_1 \leftarrow \xi_2$): method call ξ_1 must be preceded by method call ξ_2.
3) *Negative Forward Operator* ($\xi_1 \overset{-}{\longrightarrow} \xi_2$): method call ξ_1 cannot be succeeded by method call ξ_2
4) *Negative Backward Operator* ($\xi_1 \overset{-}{\longleftarrow} \xi_2$): method call ξ_1 cannot be preceded by the method call of ξ_2

We next show how ICON identifies the terms of the constraint formula:

ξ_1: 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.

ξ_2: 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.

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 $action$
```

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 findLeafContaining(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

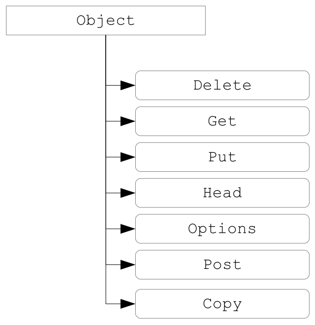


Fig. 5. Semantic Graph for the `Object` related operations in Amazon S3 REST API

actions ICON considers only the first match. For instance the sentence in Figure 3 algorithm identifies Delete Object method as ξ_2

Temporal Operator: 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 Bucket $\leftarrow$ Delete Object.

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 by Pandita et al. [21]) 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 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.

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 $seq\_Graph$
```

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 three 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 the widely used 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 any programming language.

We chose `Amazon S3`, `PayPal payment`, and `java.io` APIs as our subjects because they are widely used and contain relevant documentation.

B. Evaluation Setup

We first manually annotated the sentences in the API documents of the subject APIs. The first two authors manually labeled each sentence (2417 total) in the Java API documentation as being a temporal constraint sentence 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 evaluating effectiveness of ICON. Table III lists the subject statistics. Based on the discussions with regards to annotation of `java.io` API, first author annotated the rest of the subject APIs.

To answer RQ1, we first 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 [20], `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`. 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:

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

We then use these measures to identify relative effectiveness of ICON by executing multiple classifiers on the annotated sentences. We tested the classifiers using a stratified n-fold cross-validation approach, using 10 as the value n (10-fold) as recommended by Han et al. [12]. The cross validation ensures that every sentence is used for training and testing, thus producing low bias and variance. In particular, we execute the classifiers in two different configurations. First we executed the classifiers using the words in the sentences as features and measure `precision`, `recall`, and `F-Score` as P_{wrd}, R_{wrd}, and F_{wrd}. We next executed the classifiers on the features proposed by the ICON approach and measure `precision`, `recall`, and `F-Score` as P_{ftr}, R_{ftr}, and F_{ftr}. We next calculate relative gain in `precision`, `recall`, and `F-Score` as $P_{\triangle}$ (P_{ftr}- P_{wrd}), $R_{\triangle}$ (R_{ftr} - R_{wrd}), and $F_{\triangle}$ (F_{ftr} - F_{wrd}). Higher values of $P_{\triangle}$, $R_{\triangle}$, and $F_{\triangle}$ are indicative of effectiveness of the constraint statements inferred using the features proposed by the ICON approach.

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 the 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

TABLE III
EVALUATION RESULTS (INFERENCE)

API	Mtds	Sen	Sen_C	Spec_{ICON}	Acc(%)
java.io	662	2417	78	56	71.8
Amazon S3 REST	51	1492	12	7	58.3
Paypal REST	33	151	20	14	70.0
Total	746	4060	110	77	70.0*

* Average; Mtds: Total no. of Methods; Sen: Total no. of Sentences; Sen_C: Total no. of constraint Sentences; Acc: Accuracy Spec_{ICON}: Total no. of temporal constraint correctly identified by ICON;

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 inferring temporal constraints from the identified constraint sentences using various classifiers. Column "Trainer" lists the names of the classifiers used to train the model for classifications in ICON. Columns P_{wrd}, R_{wrd}, and F_{wrd} list the `precision`, `recall`, and `F-score` of classifiers trained without the features proposed by ICON. Columns P_{ftr}, R_{ftr}, and F_{ftr} list the `precision`, `recall`, and `F-score` of classifiers trained with the features proposed by ICON. Finally, columns $\text{P}_{\triangle}$, $\text{R}_{\triangle}$, and $\text{F}_{\triangle}$ list the improvement factors in `precision`, `recall`, and `F-score` respectively.

For our evaluation we used following well known classifiers: AdaBoost (or adaptive boosting), Naïve Bayes, Winnow, Balanced Winnow, Decision Tree, Max Entropy, and c45. We used Naïve Bayes as the weaker classifier with AdaBoost. We used `Mallet` [18] implementation of these classifiers for our experiments.

Our results show that, ICON effectively identifies constraint sentences with the average (across different classifiers) `precision`, `recall`, and `F-score` of 79.0%, 60.0%, and 65.0%, respectively. Balanced Winnow performed best, with an average `precisoin` and `recall` of 78% and 77% respectively. Furthermore, our results also show the features proposed by ICON improves the precision of classification algorithms by an average of 18%. There is also a slight increase in the recall (average 1% gain).

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 III shows the effectiveness of ICON in inferring temporal constraints from the identified constraint sentences. Column "Trainer" 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.

We next present an example to illustrate how ICON fails infer constraints from a constraint sentence. 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 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.

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

TABLE II
EVALUATION RESULTS (IDENTIFICATION)

S. No.	Trainer	P_{wrd}	R_{wrd}	F_{wrd}	P_{ftr}	R_{ftr}	F_{ftr}	$P_{\triangle}$	$R_{\triangle}$	$F_{\triangle}$
1	AdaBoost	0.52	0.81	0.61	0.7	0.78	0.74	0.18	-0.03	0.13
2	NaiveBayes	0.49	0.67	0.56	0.74	0.65	0.69	0.25	-0.02	0.13
3	Balanced Winnow	0.77	0.76	0.76	0.78	0.77	0.77	0.01	0.01	0.01
4	Decision Tree	0.82	0.58	0.66	0.92	0.41	0.56	0.1	-0.17	-0.1
5	Winnow	0.19	0.41	0.15	0.8	0.51	0.59	0.61	0.1	0.44
6	MaxEnt	0.69	0.48	0.55	0.76	0.46	0.56	0.07	-0.02	0.01
7	c45	0.78	0.4	0.52	0.82	0.59	0.67	0.04	0.19	0.15
	Average	0.61	0.59	0.54	0.79	0.6	0.65	0.18	0.01	0.11

All values are average over 10-fold cross validation; P: Precision; R: Recall; F: F-Score; $_{wrd}$: No features used for training; $_{ftr}$: features used for training; $_{\triangle}$: improvement factor ($_{ftr}$ - $_{wrd}$)

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 4000 API sentences) with the average precision, recall, and F-score of 79.0%, 60%, and 65% respectively. We also show that ICON infers temporal constraints from the constraint sentences an average accuracy of 70%. Furthermore, also provide discussion on why ICON does not or incorrectly infers temporal constraints. 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 three different API's: JDK `java.io`, `Amazon S3 REST` API, `PayPal Payment REST` API. 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 provides HTTP based access to online storage allowing developers the freedom to write clients applications in any programming language. Finally, `PayPal Payment REST` API provides a REST support for online financial transactions. The difference in the functionalities provided by the three 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 independently.

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 [26], [33], 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 [40] 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, formal temporal constraints are missing from most APIs. 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 the `PayPal Payment REST` API, the `Amazon S3 REST` API, and the commonly used package `java.io` in the JDK API. Our evaluation results show that ICON effectively identifies sentences describing temporal constraints with an average 79% precision and 60% recall, from more than 4000 sentences in subject API documents. Furthermore, ICON also achieved an accuracy of 70% in inferring 77 formal temporal constraints from these temporal constraint 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. Han and M. Kamber. *Data Mining, Southeast Asia Edition: Concepts and Techniques*. Morgan kaufmann, 2006.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] McCallum, Andrew Kachites. "MALLET: A Machine Learning for Language Toolkit.". http://mallet.cs.umass.edu.2002.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

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

[30] The Stanford Natural Language Processing Group, 1999. http://nlp.stanford.edu/.

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

[32] 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.

[33] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.