ICON: Inferring Temporal Constraints from Natural Language API Descriptions

Rahul Pandita†, Kunal Taneja‡, Teresa Tung§, and Laurie Williams†
†North Carolina State University
‡Google Inc.
§Accenture Technology Labs
rpandit@ncsu.edu, kunalltaneja@google.com, teresa.tung@accenture.com, williams@csc.ncsu.edu

Abstract—**Temporal constraints of an Application Programming Interface (API) are the allowed sequences of method invocations in the API governing the secure and robust operation of client software using the API. These constraints are typically described informally in natural language API documents, and therefore are not amenable to existing constraint-checking tools. Manually identifying and writing formal temporal constraints from API documents 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 use 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 can effectively identify temporal constraint sentences (from over 4000 human-annotated API sentences) with the average 79.0% precision and 60.0% recall. Furthermore, our evaluation demonstrates that ICON achieves an accuracy of 70% in inferring 77 formal temporal constraints from these APIs.**

I. Introduction

Application Programming Interfaces (APIs) facilitate software reuse by providing a 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 class of constraints are temporal constraints [37], which are the allowed sequences of invocations of methods from the API. Non-compliance to such constraints often results in faulty client applications that are unreliable.

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 a 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, 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.

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 the `Amazon S3 REST` API[1] spans 357 pages describing 54 methods.

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

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

For instance, consider the sentences from the `PayPal Payment REST` API [2]: 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 them. Another problem with keyword-based search is that the method names (and synonyms) are often 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, from the sentence, there is need to infer the references to the method, which often may not be explicit. Consider Sentence 2 in the preceding example. 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.

[1]http://awsdocs.s3.amazonaws.com/S3/latest/s3-api.pdf
[2]https://developer.paypal.com/docs/api/payments/

An ML-based approach for identifying the temporal constraint sentences addresses the limitations of keyword-based search by automatically learning patterns of temporal constraint sentences. We propose to use a combination of words, the syntax (parts of speech) of a sentence, and the 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 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 the natural language description of an API to infer temporal constraints of method invocations. As our approach analyzes API documents in natural language, ICON can be used independent of the programming language of an API library. Additionally, our approach complements existing mining-based approaches [2], [34], [36], [41] that partially address the problem by mining for common usage patterns among client code snippets 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 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. All our evaluation subjects, results, and artifacts are publicly available on our project website. [3]

II. Related work

We now briefly discuss the related work pertinent to our approach. In particular, we discuss the closely related approaches in the area of formal specifications, NLP in software engineering, and software documentation.

Formal Specification: Contracts, a form of formal specification, formally 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 the automated inference of contracts. Existing approaches use program analysis [5], [19], [35] to automatically infer contracts. However, studies [9], [24] 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 or properties) of a program. In contrast, temporal constraints specify the ordering of method invocations. Since ICON infers temporal constraints from API documents, we envision that ICON could work in conjunction with existing approaches to infer a comprehensive formal specification.

Another set of approaches infer contract-like specifications (such as behavioral model, algebraic specifications, and exception specifications) either dynamically [11], [13], [14] or statically [3], [9], [37] from source code and binaries. Among these approaches, Gable and Su [10] proposed to learn *micro-patterns* of temporal properties by mining runtime traces and then to combine them into larger specifications depicted as finite state automata. In contrast, ICON infers specifications from the natural language text in API documents, thus complementing existing approaches when the source code files or binaries of the API library are not available.

NLP in Software Engineering (SE): Research advances [7], [16] in the accuracy of existing NLP techniques have inspired researchers and practitioners [21]–[23], [29], [33], [39] to adapt and(/or) apply NLP techniques to solve problems in the 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 an 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 restrictions 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 a template. Furthermore, these approaches rely on specific templates for inferring constraints. In contrast, ICON identifies constraints independently of such templates.

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. Our previous work [23] proposed an NLP-based approach for inferring parameter constraints from method descriptions in the API documents. ICON differs from these approaches as follows. ICON addresses the problem of inferring temporal constraints, which is not addressed by the previous approaches. Furthermore, ICON significantly extends the infrastructure developed by in the previous work [23] in the following dimensions. First, ICON relies on ML to identify the temporal constraint sentences. The lower frequency of occurrence of temporal constraint

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

sentences in comparison to parameter constraint sentences, makes them harder to detect. Second, ICON 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 approaches rely only on parts-of-speech tags. Finally, the ICON approach leverages the concept of semantic graphs constructed from class and method names in the API to automatically infer the implicit method references in a sentence.

Augmented Documentation: Dekel and Herbsleb [8], were the first to create a tool, namely eMoose an Eclipse IDE based plug-in, that allowed developers to create directives (a 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. [8], 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.

III. Background

We next briefly introduce the NLP techniques used in this work that have been grouped into two broad categories. We first introduce the core NLP techniques, followed by the SE specific NLP techniques proposed in our previous work [22], [23].

A. Core NLP techniques

• **Parts Of Speech (POS) tagging** [15], [16]: Also known as *'word tagging'* and *'grammatical tagging'*, these techniques aim to identify the part of speech (nouns, verbs, etc.), to which a particular word in a sentence belongs.

• **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.

• **Typed Dependencies**: The Stanford typed dependencies representation [6], [7] is designed to provide a simple description of grammatical relationships directed towards non-linguistics experts to perform NLP related tasks. The representation provides a hierarchical structure for the dependencies with precise definitions of what each dependency represents, thus facilitating machine-based manipulation of natural language text.

B. SE specific NLP techniques

• **Programming Keywords** [23]: Accurate annotation of POS tags in a sentence is fundamental to the effectiveness of any advanced NLP technique. However POS tagging, initially developed for well-written news articles, often performs unsatisfactorily on domain-specific text such as API documents. In particular, with respect to API documents, certain words have a different semantic meaning, in contrast to general linguistics and this causes incorrect annotation of POS tags.

For instance, consider the word `POST`. The online Oxford dictionary has eight generic definitions of word `POST`, and none of them describes `POST` as an HTTP method[4] supported by REST API. Existing POS tagging techniques that are trained on generic documents fail to accurately annotate the POS tags of the sentences involving the word `POST`, which is likely to be noun in context of API documents in contrast to verb/noun in general linguistics.

Noun boosting identifies such words from the sentences based on a domain-specific dictionary, and annotates them appropriately. The annotation directs the POS tagger to correctly annotate the POS tags of the domain specific words thus increasing accuracy of subsequent NLP techniques such as chunking and typed dependency annotation.

• **Lexical Token Reduction** [22]: The accuracy of core NLP techniques (Section III-A) is often inversely proportional to the number of lexical tokens in a sentence. Thus, reducing the number of lexical tokens may greatly improve the accuracy of core NLP techniques. In particular, the following heuristics were proposed previously [22], [23] to achieve the desired reduction in the number of lexical tokens:

- *Named Entity Handling*: Sometimes a sequence of words corresponds to the name of entities that have a specific meaning collectively. For instance, consider the phrases *"Amazon S3" and "Amazon simple storage service"*, which are the names of the service. Further resolution of such phrases using grammatical syntax is redundant for inference needs. This heuristic annotates a phrase representing the names of an entity as a single lexical token, to improve the accuracy of core NLP techniques.
- *Abbreviation Handling*: Natural-language sentences often consist of abbreviations interleaved with text. This interleaving may cause the POS tagger to incorrectly parse the sentence. This heuristic finds such instances and annotates them as a single lexical unit. For example, text followed by an abbreviation 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** [22]: This technique accepts the syntax-annotated sentences (grammatical and semantic) and builds a First-Order-Logic (FOL) like representation of the sentence. Earlier research has shown the adequacy using FOL for NLP related analysis tasks [23], [27], [28]. Particularly, we build on the intermediate representation generator implementation of WHYPER [22]. The Intermediate-Representation Generation is implemented as a

[4]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"*

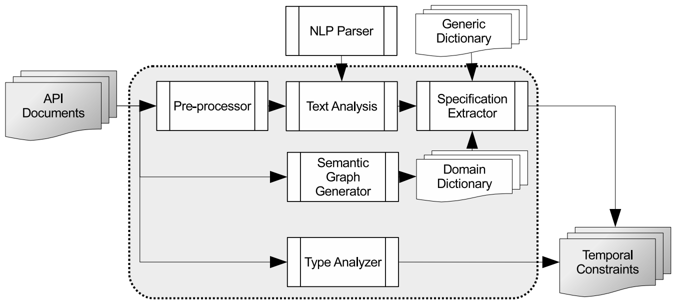

Fig. 1. Overview of the ICON approach

sequence of cascading finite state machines based on Stanford-typed dependencies [6], [7], [15], [16] annotations.

IV. ICON Overview

We next present our approach for inferring temporal constraints from the method descriptions in API Documents. Figure 1 gives an overview of the ICON approach. ICON consists of six major components: a preprocessor, a candidate identifier, a text-analysis engine, a semantic graph generator, a constraint extractor, and a type analyzer. Additionally, there is an optional model trainer component and an external NLP parser component.

First, the preprocessor accepts API documents and pre-processes the sentences in the method description. Next, an NLP parser annotates the syntax and semantics of the preprocessed sentences. The annotated sentences are accepted by the 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 sentences into the intermediate (first-order-logic like) representation. Finally, the constraint extractor leverages the semantic graphs to infer temporal constraints from the intermediate 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 detail presented to developers. ICON relies only on basic fields that are typically available for API methods across different presentation styles.

```
-> 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. 2. Sentence annotated with POS tags and Stanford-typed dependencies

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 III-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 III-B.

B. NLP Parser

The NLP parser accepts the pre-processed documents and annotates every sentence in each document using the core NLP techniques [6], [7], [22], [23], [33] described in Section III-A. In particular, each sentence is annotated with: *1) POS tags, and 2) Stanford-typed dependencies.*

Next we use an example to illustrate the annotations added by the NLP Parser. Consider the sentence from delete bucket functionality in the `Amazon S3 REST` API ***'All objects (including all object versions and Delete Markers) in the bucket must be deleted before the bucket itself can be deleted."***. Figure 2 shows the sentence annotated by the NLP parser. Each word (occurs first in black) is followed by the Part-Of-Speech (POS) tag of the word (in green), which is further followed by the name of the Stanford dependency connecting the 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, and then, using a trained ML model, classifies whether each sentence is a temporal constraint sentence 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 classify 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. The ML model can be trained offline or 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 these features.

1) Lemmatization: The features are the lemmas of a words occurring in the sentences. In linguistics, lemmatization is the reduction of the operational form of a word to its base form. For instance, "invoking", "invokes", and "invoked" are all reduced to "invoke". The rationale for using the base form of words is to reduce the size of the feature set that otherwise considers every word form as an independent feature.

2) Stop-word Reduction: In linguistics, stop words are the frequently occurring words that can be ignored and are often considered to be noise, such as "the", "of", and "to". The rationale for filtering the stop words is to reduce the size of the 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.

3) 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 following typed dependencies: "*advcl*", "*aux*", "*auxpass*", "*vmod*", and "*tmod*". For instance, the annotated sentence in Figure 2 contains both "*aux*" and "*auxpass*" dependencies. These dependencies are both added to the feature set.

4) 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, the presence or absence of determiners is unlikely to affect the outcome of classifier. We further annotate the POS of words to further distinguish between the words used in different context.

5) Sentence Structure: This 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 an ordered sequence of chunking tags is to incorporate structure of a sentence as a feature.

6) Length of a Sentence: This 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. For instance, we observed quite a few API description sentences (fragments) that were 2-3 words in length that typically describe the expanded form of camel case notation of the parameters or return value identifiers.

7) Sentence Type: This feature captures the context of the sentence, whether the sentence appears as under the method, parameter, return value, or exception/error summary as captured by the pre-processor phase. The rationale is that we observed that a majority of the temporal constraint sentences are either method or exception summary sentences.

We use the described feature set to train a classifier to perform binary classification of sentence as either temporal constraint describing or not. We use an 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 to avoid over-fitting; and 3) ML classifiers have been shown to scale well to 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 components to infer formal constraints. We define our representation as a tree structure that mimics a FOL expression. Recent research provides evidence of the adequacy of using FOL for NLP related analysis tasks [22], [23], [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. A predicate node can have at most two children. The first or the only child of a predicate node is the governing entity and the second child (if exists) is the dependent entity. Together the governing entity, predicate and the dependent entity node form a tuple.

Section III-B briefly describes the intermediate representation generation technique, that is implemented as a function of Stanford-typed dependencies [6], [7], [16], to leverage the semantic information encoded in the 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, that takes into consideration both the POS tags as well as the Stanford-typed dependencies. The POS tags that 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 intermediate representation. Thus, the intermediate representation generator used in this work is a two phase process as opposed to single phase proposed previously [22], [23]. We next describe these two phases:

- **POS Tags**: We first split a sentence into smaller constituent sentences based on the POS tags, which are then accurately annotated by the underlying NLP Parser. For instance, consider the sentence:

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

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

Fig. 3. FOL-like representation of the sentence *"All objects in the bucket must be deleted before the bucket itself can be deleted."*

The Stanford parser has trouble accurately annotating the Stanford-typed dependencies of the sentence because of presence of various clauses acting on various subject-object pairs. As shown in Figure 2 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 splits the sentence into two smaller tractable sentences, which are accurately annotated:

"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 template heuristics used in this phase. Column "Template" describes conditions where the heuristic is applicable and Column "Summary" describes the action taken by ICON when the heuristic is applicable. With respect to the previous example, Template 3 *(A noun phrase followed by another noun/pronoun/verb phrase in parenthesis)* is applicable. Thus the sentence is broken into two individual sentences.

• **Stanford-typed Dependencies**: After complex sentences have been broken to simple sentences, this phase generates an intermediate (FOL-like) representation of the sentences. This phase is equivalent to the intermediate-representation technique in proposed in WHYPER [22] as described in Section III-B. Figure 3 shows the intermediate 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 associated with a number representing the in-order traversal index of the tree.

E. Constraint Extractor

The Constraint Extractor is responsible for inferring temporal constraints from the classified constraint sentences. Temporal constraints are expressed as temporal formulae involving: a) *Predicates* ξ representing method calls and b) *Temporal operators*: backward ($\leftarrow$) & forward ($\rightarrow$) and their negations $\overset{-}{\leftarrow}$ & $\overset{-}{\rightarrow}$. We define the 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 ($\xi_1 \overset{-}{\rightarrow} \xi_2$)*: method call ξ_1 cannot be succeeded by method call ξ_2
4) *Negative Backward Operator ($\xi_1 \overset{-}{\leftarrow} \xi_2$)*: method call ξ_1 cannot be preceded by the method call of ξ_2

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

ξ_1: ICON first identifies ξ_1 as the method whose description the constraint sentence is part of. For instance the sentence in Figure 2 is part of Delete Bucket method description in `Amazon S3 REST` API, so ξ_1 is instantiated as Delete Bucket method.

ξ_2: ICON next identifies ξ_2. Since, references to ξ_2 may not always be explicit, we leverage the semantic graphs. A semantic graph is a representation of objects and the methods applicable on those objects. Figure 4 shows a graph for the `Object` resource in `Amazon S3 REST` API. The phrases in rounded rectangles are the actions applicable on an `Object` resource. Section IV-F further describes how these graphs are generated.

ICON systematically explores the intermediate representation of the candidate sentence to identify ξ_2. First, this component attempts to locate the occurrence of object name or its synonym in the leaf nodes (entity) of the intermediate representation of the sentence. Once a leaf node is found, this component systematically traverses the tree from the leaf node to the root, matching all parent predicates as well as their immediate child predicates. This component matches each of the traversed predicates with the actions associated with the object defined in the 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 the case of multiple matching actions ICON considers only the first match. For instance, for the sentence in Figure 2, 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 the reference to ξ_2 in the sentence. Past tense is considered as backward and other tenses are considered as forward. For instance, the sentence in Figure 2, since "deleted" is in past tense, the operator is backward and the constraint is Delete Bucket $\leftarrow$ Delete Object. Negation is identified by presence of the negation verbs such as "*no*", "*not*", "*can't*" ... etc. Another rule for negation is if the sentence is in the exception/error description.

F. Semantic-Graph Generator

A key way that ICON identifies a reference to a method in the API is the employment of a semantic graph of an API. We propose to initially infer such graphs from API documents. Ad hoc creation of a semantic graph is prohibitively time consuming and may be error prone. We thus employ a systematic methodology, proposed previously in Whyper [22], 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 synonym terms used to refer to the class in question. The synonym terms are listed as by

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 sentences.
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 parenthesis	Two individual sentences are formed. The first sentence is the same as the parent sentence without the noun/pronoun/verb phrase in parenthesis. The second sentence consists of of the noun phrase followed by noun/pronoun/verb phrase without the parenthesis.
4.	A noun phrase followed by a conditional phrase in parenthesis	Two individual sentences are formed. The first sentence is the same as the parent sentence without the conditional phrase in parenthesis. The second sentence consists of of noun phrases followed by the conditional in the parenthesis.
5.	A conditional phrase followed by a sentence	Two dependent sentences are formed. The first sentence consists of the conditional phrase. The second sentence consists of 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 the conjunction and the second child verb phrase. The second sentence is formulated by removing the conjunction and the first child verb phrase.

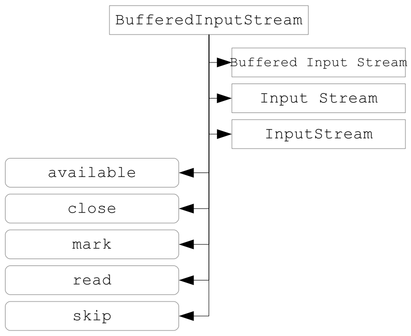

Fig. 4. Semantic Graph for the `Object` method in Amazon S3 REST API

splitting the camel-case notation in the class name. This list is further augmented with 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 each 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 a REST API we first identified the resources and then the listed REST actions on those resources as applicable actions. Figure 4 shows the graph for the `Object` resource in REST API. The phrases in rounded rectangle are the REST actions applicable on an `Object` resource in `Amazon S3 REST` API.

G. Type Analysis

Some temporal constraints are enforced by the type system in typed languages. 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). To extend the temporal constraints inferred by the analyzing the natural language text, this component infers the additional constraints that are encoded in the type system.

The component accepts the list of methods as an input and produces a graph with the nodes representing the methods in an API and the directed edges representing the temporal constraints. First, an index list (L_{mtd}) of the public methods in an API is created based on the return types of the method. Next, all public methods in the API are added as nodes ($m_1, m_2...m_k$) in an unconnected graph (G) . Next, for every method (m) in L_{mtd}, we identify the types of the input parameters ($t_1, t_2...t_k$). We then add a directed edge from all the methods in (G), where the return type of the method matches any of the types in ($t_1, t_2...t_k$). Additionally, an edge is created from the object creation methods (constructors) of a class to the non-static members methods of a class, because member methods are invoked after objects creation. The temporal constraints based on the type information area extracted by querying G. The incoming edges to a node denoting a method represents the set of prerequisite methods. The temporal constraint is then that at least one of the prerequisite methods must be invoked before invoking the method in question.

V. 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 natural language sentences describing temporal constraints? The 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? The 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 and the typed-enforced temporal constraints?

A. Subjects

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

• The `Amazon S3 REST` API provides a REST based web service 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]. We used a snapshot of the `Amazon S3 REST` API, downloaded in August 2013.

• The `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 the third quarter of 2014 alone. We used a snapshot of the `PayPal Payment REST` API, downloaded in August 2014.

• `java.io` : is one of the widely used packages in the `Java` programming language. The package provides an API for system input and output through data streams, serialization and the file system, which are among the fundamental functionalities provided by any programming language. We used a snapshot of the `java.io`, downloaded in August 2013.

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 the `cohen kappa` [4] 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 statistically significant agreement [4]. 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 the effectiveness of ICON. Table III lists the subject statistics. Based on the discussions with regards to the annotation of `java.io` API, the 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 a constraint sentence as a sentence describing a temporal constraint. 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 the ratio of number of true positives to the total number of items reported to be true, and `recall` is defined as the 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:

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

We then use these measures to identify the relative effectiveness of ICON by executing multiple machine learning 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 both 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 the 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 the constraint sentences by ICON. However, we excluded the type-enforced temporal constraints described in Section IV-G. We excluded the type-enforced constraints because they are correct by construction and are often enforced by modern IDE's such as the Eclipse by default. We then measure the *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. A higher value of `accuracy` is indicative of the 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 with the type-enforced temporal constraints inferred using the method in Section IV-G. We next present our evaluation results.

C. 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

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	0.10	-0.17	-0.10
5	Winnow	0.19	0.41	0.15	0.8	0.51	0.59	**0.61**	0.10	**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	0.40	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 ($_{ftr}$ - $_{wrd}$)

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 constraints correctly identified by ICON;

`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 $P_{\triangle}$, $R_{\triangle}$, and $F_{\triangle}$ list the improvement factors in `precision`, `recall`, and `F-score` respectively.

For our evaluation we used following well known classifiers: 1) AdaBoost (or adaptive boosting), 2) Naïve Bayes, 3) Winnow, 4) Balanced Winnow, 5) Decision Tree, 6) Max Entropy, and 7) 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 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 `precision` and `recall` of 78% and 77% respectively. Furthermore, our results also show the features proposed by ICON improve 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: Table III shows the effectiveness of ICON in inferring temporal constraints from the identified constraint sentences. Column "API" lists the names of the subject APIs. Columns "Mtds" and "Sen" list the number of methods and sentences in each subject API. 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(%)" lists the percentage values of accuracy. Our results show that, out of 110 manually identified constraint sentences, ICON correctly infers temporal constraints with an average accuracy of 70.0%.

We next present an example to illustrate how ICON may incorrectly infer 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 the documentation of the `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 to infer any constraint at all 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 the `PUT Object-Copy` method description in the `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, ICON cannot make the connection due to the limitation of the semantic graphs that do not list "already stored" as a "valid operation" on an object. In the future, we plan to investigate techniques to further improve the semantic graphs to infer such implicit constraints.

3) RQ3: Comparison to Typed-Enforced Constraints: In this section, we compare the temporal constraints inferred from the natural language API descriptions to those enforced by the type-system (referred to as type-enforced constraints). 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` semantics followed by REST operations. `CRUD` stands for the resource manipulation semantic sequence create, retrieve, update, and delete. In particular, we consider a constraint as a type-enforced constraint, if the constraint mandates that 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 inspected 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 ICON from natural language text were classified as type-enforced constraint. Hence, the constraints inferred 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 average precision, recall, and F-score of 79.0%, 60%, and 65% respectively [RQ1]. We also show that ICON infers temporal constraints from the constraint sentences with an average accuracy of 70% [RQ2]. Furthermore, we also illustrate/describe why ICON does not infer temporal constraints in some cases or infers them incorrectly. Finally, we provide a comparison of the temporal constraints inferred from natural language description against the temporal constraints enforced by a type system [RQ3].

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 APIs: JDK `java.io`, the `Amazon S3 REST` API, and the `PayPal Payment REST` API. Java is a widely used programming language and `java.io` is one of its main packages. The `Amazon S3 REST` API provides HTTP based access to online storage allowing developers the freedom to write clients applications in any programming language. Finally, the `PayPal Payment REST` API provides REST support for online financial transactions. The difference in the functionality provided by the three APIs also addresses 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 APIs.

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 from the API method descriptions in our evaluation. Furthermore, we ensured that the results were individually verified and agreed upon by two authors independently.

VI. Limitations and Future work

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

Validation of Method Descriptions. API documents can sometimes be misleading [26], [32], thus causing developers to write faulty client code. In the future, we plan to extend our approach to find documentation-implementation inconsistencies.

Inferring Implicit Constraints. The presented approach only infers temporal constraints that are explicitly described in the method descriptions. However, there are instances where the constraints are implicit. For instance, consider the method description for the `markSupported` method in the `BufferInputStream` class in Java, that states "*Test if this input stream supports* `mark`". Although trivial for a human to interpret 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.

VII. Conclusion

Despite being highly desirable, formal temporal constraints are absent in most APIs. In contrast, documentation of API methods contains detailed specifications of temporal constraints in natural language text. In this paper, we introduced a novel approach called ICON to infer temporal constraints from natural language text of API documents. We used ICON to infer these 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 achieves an accuracy of 70% in inferring 77 formal temporal constraints from these temporal constraint sentences. These results demonstrate the value in using NLP techniques to infer temporal constraints from API documents

Acknowledgment

This work is supported by the USA National Security Agency (NSA) Science of Security Lablet. Any opinions expressed in this report are those of the author(s) and do not necessarily reflect the views of the NSA. We thank Dr. Tao Xie for providing feedback and useful discussions on earlier versions this work. We also thank anonymous conference reviewers, and Realsearch research group for providing helpful feedback to improve this work.

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

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

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

[5] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proc. 30th ICSE*, pages 281–290, 2008.

[6] M. C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *Proc. LREC*, 2006.

[7] M. C. de Marneffe and C. D. Manning. The Stanford typed dependencies representation. In *Workshop COLING*, 2008.

[8] U. Dekel and J. D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.

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

[10] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proc. 16th FSE*, pages 339–349, 2008.

[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, R. P. Jetley, S. D. Sudarsan, and L. Williams. Discovering likely mappings between APIs using text mining. In *Proc. 15th IEEE International Working Conference SCAM*, pages 231–240. IEEE, 2015.

[22] 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.

[23] 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.

[24] 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.

[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. Sutton Jr., 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] 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.

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

[34] 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.

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

[36] 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.

[37] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

[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.