ICON: *I*nferring Temporal *Con*straints from Natural Language API Descriptions

Rahul Pandita[1], Kunal Taneja[2], Tao Xie[3], Laurie Williams[1], Teresa Tung[2]
[1]Department of Computer Science, North Carolina State University, Raleigh, NC, USA
[2]Accenture Technology Labs, San Jose, CA, USA
[3]Department of Computer Science, University of Illinois, Urbana-Champaign, IL, USA
rpandit@ncsu.edu, k.a.taneja@accenture.com, taoxie@illinois.edu, williams@csc.ncsu.edu, teresa.tung@accenture.com

ABSTRACT

Temporal specifications of an Application Programming Interface (API) describe allowed sequence of invocations of methods within an API. Despite being desirable for software testing and verification (through formal analysis tools), most API's do not have formal temporal specifications. In contrast, API documents contain valuable information about the usage in natural language. Although existing techniques infer from API documents method specifications (such as method precondition and postconditions) or resource specifications (such as constraints on creating/releasing resources), such techniques fail to explicitly infer temporal constraints. Manually writing formal specifications based on natural language text in API documents can be prohibitively time consuming and error prone. To address this issue, we propose a natural language processing based automated approach ICON, to infer formal temporal specifications from natural language text of API documents. ICON uses To evaluate our proposed approach, we applied ICON to infer temporal constraints from commonly used package `java.io` from JDK API and `Amazon S3 REST API`. Our evaluation results show that ICON achieves an average of 66% precision and 70% recall in inferring 69 temporal constraints from more than 3900 sentences of API documents.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; F.3.1 [**LOGICS AND MEANINGS OF PROGRAMS**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*

General Terms

Specifications

Keywords

Temporal Specifications, Natural Language Processing, API Documents

1. INTRODUCTION

Temporal specifications [3] of an Application Programming Interface (API) describe allowed sequences of invocations of methods within an API. These specifications govern the secure and robust operation of client software using these APIs. If these specifications are formal (machine-readable such as linear temporal logic), they can be used as a basis of formal analysis tools such as model checker and runtime verifiers in detecting the violations of these temporal constraints as defects.

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

One way of addressing the issue, is to manually convert the natural language API description into formal specifications. However, manually writing formal specifications based on natural language text in API documents can be prohibitively time consuming and error prone [31, 43]. For instance, Wu et al. [43] report that it took one of the authors more than 10 hours to browse the documentation of one method of a web service API, even before they attempted to formalize the constraints on the method.

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

There are existing approaches focus on inferring just the method pre-post conditions in terms of parameter constraints either using program analysis techniques [6, 14, 16–18] or using NLP [29, 43]. In contrast, temporal constraints are beyond these parameter constraints. Temporal constraints, in general focus on rules related to orchestration of methods within an API rather than focusing on the requirements on the input parameters of these methods. Our proposed approach addresses the following challenges to infer temporal constraints automatically from API documents.

First, relying just on type information will result in an incomplete list of temporal constraints. In typed languages some temporal constraints are enforced by type system. For instances a method (m) accepting input parameter (i) of type (t) mandates that (at least one) method (m') be invoked whose return value is of type (t). However, temporal constraints are not limited by the type definitions (i.e, requirements on the type of input parameters), and are currently expressed mostly in natural language in API documents, as described in the example previously. We propose to address the issue by augmenting the type definitions based temporal constraints with the constraints inferred from the natural language text in API documents.

Second, inferring temporal constraints from the natural language text itself is challenging. There are existing challenges in NLP for software engineering domain namely *ambiguity*, *programming keywords*, and *semantic equivalence* identified in previous work [29]. In addition to these challenges, we have an added challenge of identifying the method referenced in the natural language text to infer temporal constraint. Recall the description sentence "You must initiate a multipart upload before you can upload any part." In this sentence, phrases "multipart upload" and "upload any part" refer to individual methods in the API. Identifying these phrases as method invocation instances requires domain dictionaries. Ad-hoc construction of such dictionaries is prohibitively time and resources intensive. To address this challenge, we propose to build domain-dictionaries systematically from API documents and generic English dictionaries.

In summary, the proposed work leverages natural language description of API's to infer temporal constraints of method invocations. As the proposed work analyzes API documents in natural language, it can be reused independent of the programming language of the API library. Additionally, our approach complements existing mining based approaches [5, 40, 42, 45] that partially address the problem by mining for common usage patterns among client software that use the API. The proposed work in general, makes the following contributions:

- An NLP based approach that infers temporal constraints of method invocations. To the best of our knowledge, our approach is the first one to apply NLP for the goal of inferring temporal specifications from API documents.

- A prototype implementation of our approach based on extending the Stanford Parser [19, 35], 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[1].

- An evaluation of proposed approach on commonly used package `java.io` from JDK API and Amazon S3 REST API. Our evaluation results show that `ICON` achieves an average of 70% precision and 80% recall in inferring 63 temporal constraints from more than 2400 sentences of API documents.

The rest of the paper is organized as follows. Section 2 presents real world examples that motivate our approach. Section 3 discusses related work in this area. Section 4 presents the background on NLP techniques used in this work. Section 5 presents our approach. Section 6 presents evaluation of our approach. Section 7 presents a brief discussion and future work. Finally, Section 8 concludes.

[1] `https://sites.google.com/site/icon`

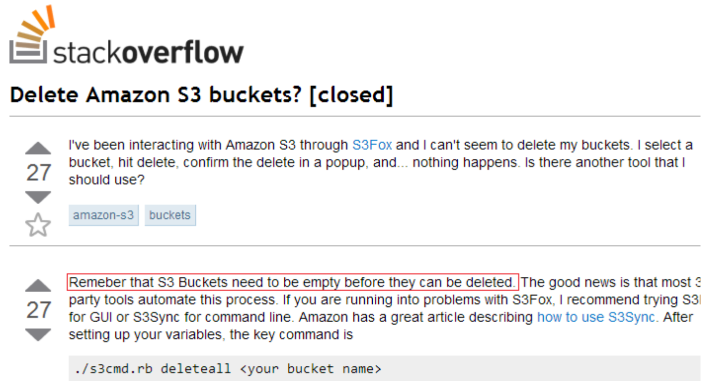


Figure 1: The Query posted on Stackoverflow form regrading Amazon S3 REST API

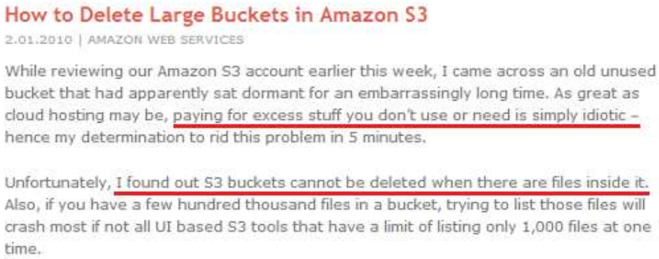
2. MOTIVATING EXAMPLE

We now present some real world examples to motivate our approach. In particular, through these examples, we demonstrate that developers often ignore the tempral specifications of an API present in its documentation. We suspect the reason for this phenomena is that API documentation is often verbose and information is distributed across various pages. For example, the PDF version of the documentation for Amazon S3 API[2] spans 278 pages. Developers do not have time and patience to go through all the documentation and miss various temporal specifications of the API, resulting in defective applications that invoke API methods in sequences prohibited by documentation. If we have the formal temporal specifications such kinds of defects can be detected by formal verification tools.

Consider the question asked in *Stack Overflow* [3] as shown in Figure 1. Stack Overflow is a question and answer site for professional and enthusiast programmers. The query is about the delete functionality of a third-party software `S3Fox` to interact with `Amazon S3 REST API`. The inquisitor complains about an issue in delete bucket functionality of the `S3Fox`. In particular, the issue was because the `S3Fox` developers overlooked the specifications in `Amazon S3 REST API` developer documentation. The document clearly 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 but notice that one of the responses on the *Stack Overflow* encouraged the person asking query to switch to another product, thus potentially resulting in a loss of revenue attributed to customer dissatisfaction. In yet another example a developer points out the financial losses he suffered because he was incorrectly using the previously described delete bucket functionality as shown in Figure 2, *("...paying for excess stuff you don't need is simply idiotic...")*.

Formal verification tools can be used to easily find such kind of defects. However, formal analysis tools are not designed to work on specifications in natural languages (such as API Documents) and require the specifications in a more formal or machine understandable format. Thus, there is need for an approach to automatically translate the constraints described in natural language into a more formal notation. In next section, we briefly discuss the related work

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



How to Delete Large Buckets in Amazon S3

2.01.2010 | AMAZON WEB SERVICES

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

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

Figure 2: The experience article posted by a developer regarding Amazon S3 REST API

in this area.

3. RELATED WORK

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

3.1 Code Contracts - Formal Specifications

Design by contracts has been an influential concept in the area of software engineering in the past decade. A significant amount of work has been done in automated inference of code contracts. There are existing approaches that statically or dynamically extract code contracts [8, 25, 41]. However, a combination of developer written and automatically inferred contracts seems to be the most effective approach [14, 30]. Furthermore, there are existing approaches that infer code-contract-like specifications (such as behavioral model, algebraic specifications, and exception specifications) either dynamically [16–18] or statically [6, 14] from source code and binaries. In contrast, the approach presented in this work infers specifications from the natural language text in API documents, thus complementing these existing approaches when the source code or binaries of the API library is not available.

3.2 NLP in Software Engineering

NLP techniques are increasingly applied in the software engineering domain. NLP techniques have been shown to be useful in requirements engineering [15, 32, 33], usability of API documents [11], and other areas [22, 28, 47]. We next describe the most relevant approaches.

1. Access Control Policies: Xiao et al. [44] and Slankas et al. [34] use shallow parsing techniques to infer Access Control Policy (ACP) rules from natural language text in use cases. The use of shallow parsing techniques works well on natural language texts in use cases, owing to well formed structure of sentences in use case descriptions. In contrast, often the sentences in API documents are not well formed. Additionally, these approaches do not deal with programming keywords or identifiers, which are often mixed within the method descriptions in API documents.

2. Resource Specifications: Zhong et al. [46] employ NLP and Machine Learning (ML) techniques to infer resource specifications from API documents. Their approach uses machine learning to automatically classify such rules. In contrast, we attempt to parse sentences based on semantic templates and demonstrate that such an approach preforms reasonably well. Furthermore, the performance of the proposed approach is dependent on the quality of the training sets used for ML. In contrast, approach presented in this work is independent of such training set and thus can be easily extended to target respective problems addressed by them.

3. Code Comments: Tan et al. [36] and Hwei-Tan et al. [38] applied an NLP and ML based approach on code comments/ Javadoc comments to detect mismatches between these comments and implementations. They rely on predefined rule templates targeted towards method invocation and lock related comments, thus limiting their scope both in terms of application area as well as language used in the comments. In contrast, approach presented in this report relies on generic natural language based templates thus relaxing the restriction on the style of the language used to describe specifications.

4. API Description: Most closely related work to the approach presented here is Pandita et al. [29] work on inferring parameter constraints from method descriptions in the API documents. In contrast this approach deals with the temporal constraints. The approach presented here is a significant extension to the infrastructure used in the previous work as documented in Section 5.

3.3 Augmented Documentation

Another related field has been that of improving the documentation related to an software API [11, 37]. We next describe most relevant approaches. Dekel et al. [11], 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. [21] improved upon their work by providing a formalism to the directives proposed by Dekel et al. [11], thus allowing tool based verification. However, a developer has to manually annotate such directives. In contrast, our proposed approach both identifies the sentences pertaining to temporal constraints and infers the temporal constraints automatically.

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

4. BACKGROUND

Natural language is well suited for human communication, but converting natural language into unambiguous specifications that can be processed and understood by computers is difficult. However, research advances [9,10,19,20] have increased the accuracy of existing NLP techniques to annotate the grammatical structure of a sentence. These advances in NLP have inspired researchers/practitioners [28, 29, 34, 39, 44] to adapt/apply NLP techniques to solve problems in software engineering domain.

In particular, this work proposes novel techniques on top of previously proposed techniques [28, 29] to demonstrate the effectiveness of applying NLP on API documents. We next briefly introduce the techniques used in this work that have been grouped into broad categories. We first introduce the core NLP techniques used in this work. We then introduce the software engineering specific NLP techniques proposed in previous work [28, 29] that are used in this work.

4.1 Core NLP techniques

Parts Of Speech (POS) tagging [19, 20]. 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

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

demonstrated to achieve 97% [35] accuracy in classifying POS tags for well written news articles.

Phrase and Clause Parsing. Also known as chunking, this technique divides a sentence into a constituent set of words (or phrases) that logically belong together (such as a Noun Phrase and Verb Phrase). Chunking thus further enhances the syntax of a sentence on top of POS tagging. Current state-of-the-art approaches can achieve around 90% [35] accuracy in classifying phrases and clauses over well written news articles.

Typed Dependencies [9, 10]. The Stanford typed dependencies representation is designed to provide a simple description of grammatical relationships directed towards non-linguistics experts to perform NLP related tasks. It provides a hierarchical structure for the dependencies with precise definitions of what each dependency means, thus facilitating machine based manipulation of natural language text.

Named Entity Recognition [13]. Also known as *'entity identification'* and *'entity extraction'*, these techniques are a subtask of information extraction that aims to classify words in a sentence into predefined categories such as names, quantities, expression of times, etc. These techniques help in associating predefined semantic meaning to a word or a group of words (phrase), thus facilitating semantic processing of named entities.

4.2 Software engineering specific NLP techniques

Noun Boosting [29]. Accurate annotation of POS tags in a sentence is fundamental to effectiveness of any advanced NLP technique. However, as mentioned earlier, POS tagging works well on well written news articles which does not necessary entail that the tagging works well 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 very different semantic meaning, in contrast to general linguistics that causes incorrect annotation of POS tags.

Consider the word `POST` for instance. The online Oxford dictionary[5] has 8 different definition of word `POST`, and none of them describes `POST` as a 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 in turn increasing accuracy of advanced NLP techniques such as chunking and typed dependency annotation.

Lexical Token Reduction [28]. These are a group of generic preprocessing heuristics to further improve the accuracy of core NLP techniques. The accuracy of core NLP techniques is inversely proportional to the number of lexical tokens in a sentence. Thus, the reduction in the number of lexical tokens greatly increases the accuracy of core NLP techniques. In particular, following heuristics have been used in previous work to achieve the desired reduction of lexical tokens:

- **Period Handling**. Besides marking the end of a sentence in simplistic English, the character period ('.') has other legal usages as well such as decimal representation (periods between numbers). Although legal, such usage hinder detection of sentence boundaries, thus causing core NLP techniques to return incorrect or imprecise results. The text is pre-processed by annotating these usages for accurate detection of sentence boundaries.

- **Named Entity Handling**. Sometimes a sequence of words correspond to the name of entities that have a specific meaning collectively. For instance, consider the phrases *"Amazon S3"*, *"Amazon simple storage service"*, which are the names of the service. Further resolution of these phrases using grammatical syntax is unnecessary and would not bring forth any semantic value. Also these phrases contribute to length of a sentence that in turn negatively affects the accuracy of core NLP techniques. This heuristic annotates the phrase representing the name of the entities as a single lexical token.

- **Abbreviation Handling**. Natural-language sentences often consist of abbreviations mixed with text. This phenomenon can result in subsequent components to incorrectly parse a sentence. This heuristic finds such instances and annotates them as a single lexical unit. For example, text followed by abbreviations such as *"Access Control Lists (ACL)"* is treated as single lexical unit. Detecting such abbreviations is achieved by using the common structure of abbreviations and encoding such structures into regular expressions. Typically, regular expressions provide a reasonable approximation for handling abbreviations.

Intermediate-Representation Generation [28]. This technique accepts the syntax-annotated sentences and builds a First-Order-Logic(FOL) representation of the sentence. Earlier researches have shown the adequacy using FOL for NLP related analysis tasks [29, 32, 33]. In particular, WHYPER [28] demonstrates the effectiveness of this technique, by constructing an intermediate representation generator based on shallow parsing [4] techniques. The shallow parser itself is implemented as a sequence of cascading finite state machines based on the functions of stanford-typed dependencies [9, 10, 19, 20].

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

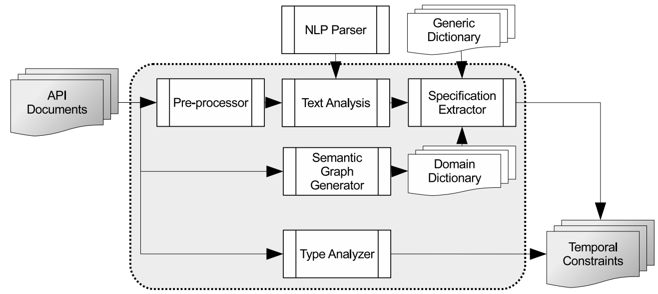


Figure 3: Overview of ICON approach

5. ICON DESIGN

We next present our approach for inferring specifications from the method descriptions in API Documents. Figure 3 gives an overview of our approach. Our approach consists of five major components: a preprocessor, a text-analysis engine, a semantic graph generator, specification extractor, and a type analyzer.

[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"*

The preprocessor accepts API documents and preprocesses the sentences in the method description, such as annotating sentence boundaries and reducing lexical tokens. The text-analysis engine accepts the pre-processed sentences and annotates them using an NLP parser. The text-analysis engine further transforms the annotated sentences into the first-order-logic (FOL) representation. The semantic graph generator accepts the API documents and generates the semantic graphs that are leveraged by specification extractor component. The type analyzer component infers temporal constraints encoded in the type system of a language by analyzing the API methods parameter and return types. Finally, the specification extractor then leverages the semantic graphs to infer temporal constraints from the FOL representation of a sentence. We next describe each component in detail.

5.1 Preprocessor

The preprocessor accepts the API documents and first extracts method descriptions from it. In particular, the preprocessor extracts the following fields within 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 by the methods*.

This step is required to extract the desired descriptive text from the various presentation styles of the API documents. In particular, different API documents may have different styles of presenting information to developers. This difference in style may include the difference in the level of detail presented to the developer. Our approach thus relies on only basic fields that are trivially 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 4.1) that are used in the subsequent phases of ICON approach. In particular, the preprocessor first employs the noun boosting followed by heuristics listed under lexical token reduction, as introduced in Section 4.2.

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

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

Prototype Implementation. Currently our prototype implementation works with online `Amazon S3 REST API` developer documentation and JDK API. However, almost all of the developer documents are provided online as structured webpages. Thus, current implementation of preprocessor can be easily extended to extract the desired information from any API developer documents.

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

Finally, to achieve n-gram reduction we used Apache Lucene® [23]. Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.

5.2 NLP Parser

The NLP parser accepts the pre-processed documents and annotates every sentence within each document using core NLP techniques described in Section 4.1. From an implementation perspective, we chose the Stanford parser [24]. However, this component can be implemented using any other existing NLP libraries or approachs. In particular, we annotate each sentence with POS tags, named-entity annotations and Stanford-typed dependencies. For more details on these techniques and their application, please refer to [9, 10, 28, 29, 39].

5.3 Text Analysis Engine

The text analysis engine component accepts the annotated documents and creates an intermediate representation of each sentence. We define our representation as a tree structure that is essentially a FOL expression. Research literature provides evidence of the adequacy of using FOL for NLP related analysis tasks [28, 29, 32, 33].

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 4.2 the intermediate representation generation technique is based on the principle of shallow parsing [4]. In particular, the intermediate-representation technique is implemented as a function of Stanford-typed dependencies [9, 10, 20], to leverage the semantic information encoded in Stanford-typed dependencies.

However, we observed that such implementation is overwhelmed by complex sentences. This limitation mandates the use of additional novel technique of *'Frequent Phrases Reduction'* in preprocessing phase. We further improve the accuracy of intermediate-representation generation by proposing 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 [28, 29]. We next describe these two phases:

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

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

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

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

Table 1 shows a list the semantic templates used in this phase. Column "Template" describes conditions where the template is applicable and Column "Summary" describes the action taken by our shallow parser when the template is applicable. All of these semantic templates are publicly available on our project website [1]. With respect to the previous example the template 3 *(A noun phrase followed by another noun/pronoun/verb phrase in brackets)* is applicable. Thus our shallow parser breaks the sentence into two individual sentences.

Stanford-typed Dependencies: This phase is equivalent to the intermediate-representation technique described in Section 4.2.

5.4 Specification Extractor

This component accepts the FOL representation of the sentence from the previous component, then extracts the temporal constraints if present in a sentence. Specification Extractor then classifies the sentence as a specification sentence (containing temporal constraint) candidate based on following ordered set of rules:

1. The sentence is not from parameter summary or return variable summary. Typically such sentences describe pre-post conditions as opposed to temporal constraints this approach addresses.

2. The sentences contains modal modifiers such as "*can, could, may, must, should*" expressing necessity. Typically, presence of such modal modifier is a strong indicator of presence of constraints imposed by an API developer

3. If the sentence does not contain modal modifiers described previously, sentence must contain temporal modifier relationship, identified by Stanford-typed dependency parser. Typically, presence of temporal modifier is an indicator of presence of temporal information.

4. If rules 2 and 3 don't apply then the sentences should be a conditional sentence, identified by the presence of keywords such as "*if*".

Once a candidate sentence is identified, this component selects an semantic graph. In particular, the semantic graph of the API class to which the candidate sentence belongs to selected. A semantic graph constitutes the keyword representation of the classes and the corresponding applicable actions. Figure 4 shows a sub-graph of graph for `BufferedInputReader` class. The phrases in solid rectangles are synonyms of the class name `BufferedInputReader`. The phrases in rounded rectangle are the actions applicable on `BufferedInputReader` class. Section 5.5 further describes how these graphs are generated.

Specification extractor then uses the semantic graph to determine if a candidate sentence is a specification sentence and if so extract the action that should be performed prior to the method the sentence belongs to. Algorithm 1 describes this action extraction process.

Our algorithm systematically explores the FOL representation of the candidate sentence to determine if a sentence describes a temporal specification. First, our algorithm attempts to locate the occurrence of class name or its synonym within the leaf nodes of the

Algorithm 1 Action_Extractor

```
Input:  K_Graph g, FOL_rep rep
Output: String action
 1: String action = φ
 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
```

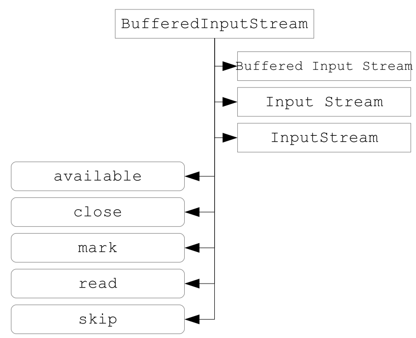


Figure 4: Semantic Graph for the `BufferedInputStream` class in Java

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 class name or one of its synonyms. In particular, we use WordNet [12] and Lemmatisation to deal with synonyms of a word in question to find appropriate matches. 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].

Our algorithm matches each of the traversed predicate with the actions associated with the class defined in semantic graph. Similar to matching entities, we also employ WordNet and Lemmatisation to deal with synonyms to find appropriate matches. If a match is found, then the matching action name is returned.

5.5 Semantic-Graph Generator

A key way of identifying reference to a method within the API in our proposed approach is the employment of a semantic graph of an API. In particular, we propose to initially infer such graphs from API documents. Manually creating a semantic graph is prohibitively time consuming and may be error prone. We thus employ a systematic methodology (proposed previously in [28]) to infer such semantic graphs from API documents that can potentially be automated. 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

Table 1: 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.

actions applicable to the objects represented by the class. From the name of a public method (describing a possible action on the object), we extract verb phrases. The verb phrases are used as the associated actions applicable on the object. For instance, `BufferedInputReader` defines operations available, close, mark, and so on. We associate these operations with the objects of type `BufferedInputReader`. Figure 4 shows a sub-graph of graph for `BufferedInputReader` class. The phrases in solid rectangles are synonyms of the class name `BufferedInputReader`. The phrases in rounded rectangle are the actions applicable on `BufferedInputReader` class.

5.6 Type Analysis

As mentioned earlier that some temporal constraints are enforced by the type system in typed Languages. For instances a method (m) accepting input parameter (i) of type (t) mandates that (at least one) method (m') be invoked whose return value is of type (t). To extend the temporal constraints inferred by the analyzing the natural language text, this component infers additional constraints that are encoded in the type system. Algorithm 2 lists the steps followed to infer type based temporal constraints.

The algorithm accepts the list of methods as an input produces a graph with the nodes representing methods in an API and the directed edges representing temporal constraints. First, a 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.

Algorithm 2 Type_Sequence_Builder

```
Input: List methodList
Output: Graph seq_Graph
 1: Graph seq_Graph = φ
 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
```

6. EVALUATION

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

- **RQ1**: What are the precision and recall of ICON in identifying temporal constraints from sentences written in natural language?
- **RQ2**: What is the accuracy of ICON in inferring temporal constraints from specification sentences in the API documents?
- **RQ3**: How do the constraints inferred by our approach compare with the typed-enforced temporal constraints?

6.1 Subjects

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

- Amazon S3. `Amazon S3 REST API` developer documentation provides a REST based web services interface that can be used to store and retrieve data on the web. Furthermore, `Amazon S3` also empowers a developer with rich set of API methods to access a highly scalable, reliable, secure, fast, inexpensive infrastructure. `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 [2].

- java.io. `java.io` is a popular packacge in `Java` programming language that provides APIs for system input and output through data streams, serialization and the file system.

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

6.2 Experimental Setup.

We first manually annotated the sentences in the API documents of the two APIs. Two authors manually labeled each sentence in the API documentation as sentence containing temporal constraints or not. We used `cohen kappa` [7] 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. After the authors classified all the sentences, they discussed with each other to reach a consensus on the sentences they classified differently. We use this classified sentences as the golden set for calculating precision and recall.

To answer RQ1, we measure the number of true positives (TP), false positives (FP), true negative (TN), and false negatives (FN) in identifying the specification sentences by ICON. We define specification 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 specification sentence.
2. FP: A sentence incorrectly identified by ICONas specification sentence.
3. TN: A sentence correctly identified by ICON as not a specification sentence.
4. FN: A sentence incorrectly identified by ICON as not a specification sentence.

In statistical classification [27], $Precision$ is defined as a ratio of number of true positives to the total number of items reported to be true, $Recall$ is defined as a ratio of number of true positives to the total number of items that are true. $F - score$ is defined as the weighted harmonic mean of $Precision$ and $Recall$. Higher value of $Precision$, $Recall$, and $F - Score$ are indicative of higher quality of the specification statements inferred using ICON. based on the calculation of TP, FP, TN, and FN of ICON defined previously we computed the $Precision$, $Recall$, and $F - Score$ of ICON as follows:

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

To answer RQ2, we checked the temporal constraints inferred from specification sentences by ICON. We measure $accuracy$ of ICONas the ratio of the total number of temporal constraints that are correctly inferred by ICONto the total number of specification sentences.

6.3 Results

We next describe our evaluation results to demonstrate the effectiveness of ICON in identifying temporal constraints.

6.3.1 *RQ1: Effectiveness in Identifying Specification Sentences*

In this section, we quantify the effectiveness of ICON in identifying specification sentences by answering RQ1. Table 2 shows the effectiveness of ICON in identifying specification sentences. Column “API” lists the names of the subject API. Column “Methods” and “Sentences” lists the number of methods and sentences in each subject API’s. Column “ICON Ann” lists the number of sentences identified by ICON as constraint sentences. Columns “TP”, “FP”, “TN”, and “FN” represent the number of `true positives`, `false positives`, `true negatives`, and `false negatives`, respectively. Columns “P(%)”, “R(%)”, and “F-Score(%)” list percentage values of `precision`, `recall`, and `F-score` respectively. Our results show that, out of 3,909 sentences, ICON effectively identifies specification sentences with the average precision, recall, and F-score of 65.0%, 72.2%, and 68.4%, respectively.

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

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

If we manually fixed the Wordnet dictionaries to match “has” and “get” as synonyms, our precision is further increased to 70.8% effectively increasing the F-Score of ICONto 71.2%. We refrained from including such modifications for reporting the results to stay true to our proposed framework. In the future, we plan to investigate techniques to construct better domain dictionaries for software API.

We next present an example to illustrate how ICON fails identify a specification sentence (producing false negative). False negatives are particularly undesirable in the context of our problem domain, because they can mislead the users of ICON into believing that no other temporal constraint exists in the API document. Furthermore, an overwhelming number of false negatives works against the usefulness of ICON. For instance, consider the sentence “*This is done by flushing the stream and then closing the underlying output stream.*” from `close` method description from `PrintStream` class. The sentence describes the constraint in terms of invoking the same method again. Although ICON correctly identifies the method `close` in the sentence, ICON currently does not support temporal constraint on the same method. Aforementioned, limitation causes ICON to not identify the statement as a specification constraint.

Another major source of false negatives (similar to reasons for

Table 2: Evaluation Results

API	Methods	Sentences	Spec Sentences	ICON Ann	TP	FP	FN	P(%)	R(%)	F-Score(%)
java.io		2417	78	88	57	31	21	64.8	73.1	68.8
AMAZON S3 REST	51	1492	12	12	8	4	4	66.7	66.7	66.7
Total		3909	90	100	65	35	25	65.0*	72.2*	68.4*

* Column average; TP: Total number of True Positives; FP: Total number of False Positives; FN: Total number of False Negatives; P: Precision; R: Recall

false positives) is the incorrect parsing of sentences by the underlying NLP infrastructure. For instance, consider the sentence "*If any in-memory buffering is being done by the application (for example, by a BufferedOutputStream object), those buffers must be flushed into the FileDescriptor (for example, by invoking OutputStream.flush) before that data will be affected by sync.*" The sentence describes that the `OutputStream.flush()` must be invoked before invoking the current method if in-memory buffering is performed. However, the length and complexity in terms of number of clauses causes the underlying Stanford parser to inaccurately annotate the dependencies, which eventually results into incorrect classification.

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

6.3.2 RQ2: Accuracy in Inferring Temporal Constraints

In this section, we evaluate the effectiveness of ICON in inferring specification sentences from API documents.

6.3.3 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. For answering this question we used the constraints inferred by the Algorithm 2 presented in Section 5 on classes in `java.io` package.

We observed that the constraints inferred by our ICON from natural language text were distinct from the constraints enforced by the type system.

6.4 Summary

6.5 Threats to Validity

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

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

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

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

7. DISCUSSION AND FUTURE WORK

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

7.1 Limitations:

Leveraging Error Descriptions

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

Semantic Flow

8. CONCLUSION

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

9. REFERENCES

[1] Whyper. https://sites.google.com/site/whypermission/.

[2] 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-html`.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

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

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

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

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

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

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

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

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

[12] F. et al. *WordNet: an electronic lexical database.* Cambridge, Mass: MIT Press, 1998.

[13] J. R. Finkel, T. Grenager, and C. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proc, 43nd ACL*, 2005.

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

[15] V. Gervasi and D. Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transactions Software Engineering Methodologies*, 14:277–330, 2005.

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

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

[18] 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.

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

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

[21] 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.

[22] G. Little and R. C. Miller. Keyword programming in Java. In *Proc. 22nd ASE*, pages 84–93, 2007.

[23] Apache Lucene Core. `http://lucene.apache.org/core/`.

[24] C. Manning and H. Schutze. Foundations of statistical natural language processing. *The MIT Press*, 2001.

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

[26] D. G. Novick and K. Ward. Why don't people read the manual? In *Proc. 24th ACM*, pages 11–18, 2006.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

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

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

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

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

[37] 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.

[38] 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.

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

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] H. Zhou, F. Chen, and H. Yang. Developing Application

Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology. In *Proc. 8th QSIC*, pages 225 –234, 2008.