I♡CON: *I*nferring ♡*L*ikely usage *Con*straints from REST developer documents

***Abstract*—Recently many popular websites such as Twitter and Flickr expose their data through web service APIs, enabling third-party organizations to develop client applications that provide functionalities beyond what the original websites offer. These client applications should follow certain constraints in order to correctly interact with the web services. Violations of such constraints can cause fatal errors or incorrect results in the client applications. However, these constraints are often not formally specified and thus not available for automatic verification of client applications. To address this issue, we propose a novel approach, called I♡CON,**

I. INTRODUCTION

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

II. BACKGROUND

Although, well suited for human communication, converting natural language into unambiguous specifications that can be processed and understood by computers is very difficult. Recently, a lot of exciting work has been carried out in the area of Natural Language Processing (NLP), with existing NLP techniques proving to be fairly accurate in highlighting grammatical structure of a natural language sentence. However, existing NLP techniques are still in the processing phase and not in understanding phase. We briefly introduce the NLP techniques used in this work.

Parts Of Speech (POS) tagging [22], [23]. 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 been have demonstrated to achieve 97% [36] 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% [36] accuracy

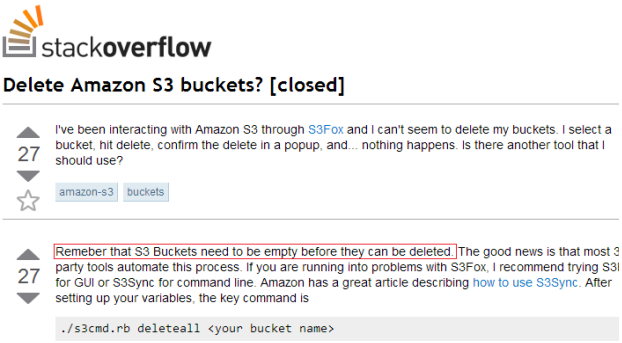

Figure 1. The Query posted on Stackoverflow form regrading Amazon S3 REST API

in classifying phrases and clauses over well written news articles.

Typed Dependencies [6], [7]. 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 [12]. Also known as *'entity identification'* and *'entity extraction'*, these techniques are a subtask of IE that aims to classify words in a sentence into predefined categories such as names, quantities, expression of times, etc. These techniques help in associating predefined semantic meaning to a word or a group of words (phrase), thus facilitating semantic processing of named entities.

Co-reference Resolution [24], [32]. Also known as *'anaphora resolution'*, these techniques aim to identify multiple expressions present across (or within) the sentences, that point out to the same thing or *'referant'*. These techniques are useful for extracting information; especially if the information encompasses many sentences in a document.

III. EXAMPLES

IV. I♡CON DESIGN

We next present our framework for inferring specifications from the method descriptions in API Documents. Figure 3 gives an overview of our framework. Our framework consists of four major components: a parser, a preprocessor, an text-analysis engine, and a postprocessor.

DELETE Bucket

Description

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

Requests

Syntax

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

Request Parameters

This implementation of the operation does not use request parameters.

Request Headers

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

Request Elements

This implementation of the operation does not use request elements.

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

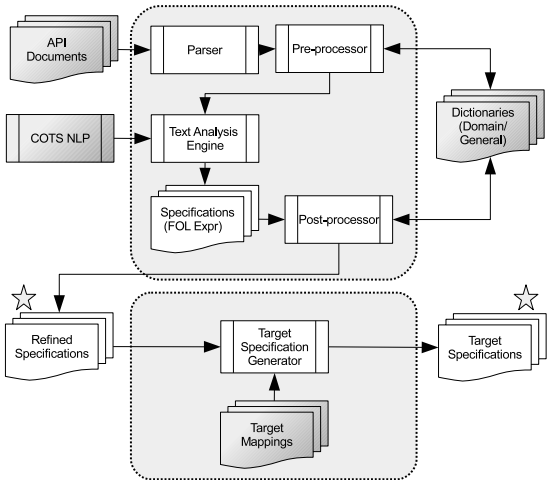

Figure 3. Overview of I♡CON framework

The parser accepts online API description and parser them into an intermediate representation. The pre-processor accepts method descriptions and preprocesses the sentences in the descriptions, such as annotating sentence boundaries and reducing lexical tokens. The text-analysis engine accepts the pre-processed sentences and parses them using an NLP parser. The text-analysis engine further transforms the parsed sentences into the first-order-logic (FOL) representation and then leverages the FOL representation of a sentence to infer specifications. Finally, the postprocessor undoes any modifications done by the preprocessor to get refined specifications.

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

A. Parser

Our parser accepts the online API documents and parses them into predefined categories. In particular, our parser extracts the following contents: 1) Summary of the API method, 2)Summary of request elements of the method, and 3) Summary of response elements of the method inclusive of error responses.

This step is required to extract the desired descriptive text from the various presentation styles of the API documents. In particular, deferential API documents may have styles of presenting information to developers inclusive of different levels of details, we chose very generic fields that are found across all documents.

Currently our prototype implementation extracts this information from `Amazon S3 REST API` developer documentation. However almost all of the REST API developer documents are provided on-line as structured webpages, thus the parser can be easily extended to extract desired information from any REST API developer documents in general.

B. Preprocessor

The preprocessor accepts method application descriptions extracted by the parser and preprocesses the natural language text, to be further analyzed by the text-analysis engine. In particular, the preprocessor annotates special words and reduces the number of lexical tokens using semantic information. The preprocessing steps are required to increase the accuracy of the subsequent phases of I♡CON framework, described in detail in Section IV-D. We next describe in detail the preprocessing steps:

1. Special word annotation. Since the exiting NLP techniques rely on accurate annotation of POS tags in a sentence, the accuracy of I♡CON to infer constraints is dependent on the accurate annotation of POS tags. While existing techniques work well on general linguistics, that does not necessary entail that they work well with domain specific text as well. 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 [1] has 8 different definition of word "POST", and none of them describes POST as a HTTP method supported by REST API meaning: "*Creates a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation*". Thus existing NLP techniques fail to accurately annotate the POS tags of the sentences involving word "POST".

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

This preprocessing step identifies such words from the sentences based on a domain-specific dictionary, and annotates them appropriately. This special annotation forces the underlying POS tagger to accurately annotate the POS tags of the words involving domain specific keywords.

From implementations perspective, we manually built the domain-specific dictionary using the glossary of terms collected from the web pertaining to REST API. We further leveraged the HTML style information in `Amazon S3 REST API` developer documentation to look for words that were highlighted in code like format.

2. Lexical tokens reduction.

The reduction of lexical tokens greatly increases the accuracy of the analysis in the subsequent components of our framework. In particular, the preprocessor reduces lexical tokens by performing following subtasks

1. 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 forcing the subsequent components to return incorrect or imprecise results. We pre-process the sentences by annotating these usages for accurate detection of sentence boundaries. We achieve so by looking up known shorthand words from WordNet [11] and detecting decimals, which are also the period character, by using regular expressions. From an implementation perspective, we have maintained a static lookup table of shorthand words observed in WordNet.

2. 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 POS tagger. Thus, we identify such phrases and annotate them as single lexical units. We achieve so by maintaining a static lookup table.

3. Abbreviation Handling. Natural-language sentences often consist of abbreviations mixed with text. This can result in subsequent components to incorrectly parse a sentence. We find such instances and annotate them as a single entity. 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.

4. Frequent Phrases Although the previous steps assist in bringing down the number of lexical tokens in a sentence, some sentences may still contain a considerable number of lexical tokens to overwhelm the underlying POS tagger. To address this issue we propose to further reduce the number of lexical tokens in a sentence by annotating frequent phrases as a single lexical unit.

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

From an implementation perspective we used Apache Lucene® [26] to achieve n-gram reduction. Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.

Although a POS tagger can be retrained to achieve these pre-processing steps, we prefer annotations to make our approach independent of any specific NLP infrastructure, thus ensuring interoperability with various POS taggers.

C. NLP Parser

The NLP parser accepts the pre-processed documents and annotates every sentence within each document using standard NLP techniques. From an implementation perspective, we chose the Stanford Parser [21]. However, this component can be implemented using any other existing NLP libraries or frameworks. In particular, we annotate each sentence with POS annotations, Named-Entity Annotations and Stanford-Typed Dependencies. For more details on these tehniques and their application, please refer to [6], [7], [29], [30].

D. Text Analysis Engine

The text analysis engine component accepts the annotated documents and creates an intermediate representation of each sentence. We define our representation as a tree structure that is essentially a First-Order-Logic (FOL) expression. Research literature provides evidence of the the adequacy of using FOL for NLP related analysis tasks [29], [30], [34], [35].

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.

In particular, the process of generation of intermediate-representation is an extension of the intermediate-representation generator component of *WHYPER* [29] with additional steps. In *WHYPER*, the intermediate representation is generated based on principle of shallow parsing [3]. They implemented their parser as a function of Stanford-typed dependencies [6], [7], [21], [23], to leverage the semantic information encoded in Stanford-typed dependencies.

However, we observed that intermediate-representation generator component of *WHYPER* [29] is overwhelmed by wordy sentences.This limitation mandates the use of additional novel technique of *'Frequent Phrases Reduction'* in preprocessing phase. We further improve the accuracy of parser by adopting a hybrid (leveraging both POS tags as well as Stanford-typed dependencies) approach.

Our implementation of shallow parser is a two phase process:

1) **POS Tags**: We first parse a sentence based on the function of POS tags. In particular, we use semantic templates to logically break a sentences into smaller constituent sentences. For instance, consider the sentence "All objects (including all object versions and Delete Markers) in the bucket must be deleted before the bucket itself can be deleted.". The Stanford parser inaccurately annotates the Stanford-typed dependencies of the sentence because of presence of different clauses acting on different subject-object pairs. We thus break down the sentence into two smaller tractable sentences:
   All objects in the bucket must be deleted before the bucket itself can be deleted.
   All objects including all object versions and Delete Markers.
   Table I shows a list the semantic templates. Column "Template" describes conditions where the template is applicable and Column "Summary" describes the action taken by our shallow parser when the template is applicable. All of these semantic templates are publicly available on our project website [1]. With respect to the previous example the template no. 3 *(A noun phrase followed by another noun/pronoun/verb phrase in brackets)* is applicable. Thus our shallow parser breaks the sentence into two individual sentences.
2) **Stanford-typed Dependencies**: This phase is equivalent to the intermediate-representation generator component of the *WHYPER* [29].

V. Evaluation

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

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

A. Subjects

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

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

TBD.

B. Summary

C. Threats to Validity

Threats to external validity primarily include the degree to which the subject documents used in our evaluations are representative of true practice. To minimize the threat, we used API documents of two representative commercial REST API: one dealing with online storage and the other **TBD**. The `Amazon S3 REST API` developer 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.

VI. Discussion and Future work

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

A. Limitations:

Leveraging Error Descriptions

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

Semantic Flow

Table I
SEMANTIC TEMPLATES

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

VII. RELATED WORK

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 [5], [28], [39]. However, a combination of developer written and automatically inferred contracts seems to be the most effective approach [13], [31]. Since developers describe the specifications in the method descriptions, we believe that our approach can work in conjunction with existing approaches towards extracting a comprehensive set of code contracts for a method. Furthermore, Wei et al. [42]demonstrated that dynamic contract inference performed better when provided with an initial set of seed contracts.

There are existing approaches that infer code-contract-like specifications (such as behavioral model, algebraic specifications, and exception specifications) either dynamically [15], [19], [20] or statically [4], [13] from source code and binaries. In contrast, our approach 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.

NLP techniques are increasingly applied in the software engineering domain. NLP techniques have been shown to be useful in requirements engineering [14], [34], [35], usability of API documents [8], and other areas [25], [48]. We next describe most relevant approaches.

Xiao et al. [43] 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 nature of sentences in use case descriptions. In contrast, often the sentences in API documents are not well formed. Additionally, their approach does not deal with programming keywords or identifiers, which are often mixed within the method descriptions in API documents.

[30] [29]

Zhong et al. [47] employ NLP and 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. Tan et al. [38] applied an NLP and Machine Learning (ML) based approach to test Javadoc comments against implementations. However, their approach specifically focuses on null values and related exceptions, thus limiting the application scope. In contrast, our approach infers generic specifications from API documents. In particular, our approach already produces FOL representation of the specifications that can be used to test implementations. Furthermore, the performance of the preceding ML-based approaches is dependent on the quality of the training sets used for ML. In contrast, our approach is independent of such training set and thus can be easily extended to target respective problems addressed by these approaches.

Among other works described in [33], Mining API Mapping (MAM) [46] is most directly related to our work. MAM mines API mapping relations across different languages for language migration, however they do little in terms of mining usage constraints of methods.

VIII. CONCLUSION

REFERENCES

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

[2] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proc. 19th CCS*, pages 217–228, 2012.

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

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

[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] Q. Do, D. Roth, M. Sammons, Y. Tu, and V. Vydiswaran. Robust, Light-weight Approaches to compute Lexical Similarity. Computer science research and technical reports, University of Illinois, 2009.

[10] M. El-Ramly, R. Eltayeb, and H. Alla. An experiment in automatic conversion of legacy Java programs to C#. In *Proc. IEEE CSA*, pages 1037–1045, 2006.

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

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

[13] W. Frakes and K. Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529 – 536, 2005.

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

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

[16] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proc. 35nd ICSE*, 2013.

[17] G. Gregory. *Light Parsing as Finite State Filtering.* Cambridge University Press, 1999.

[18] A. E. Hassan and R. C. Holt. A lightweight approach for migrating web frameworks. *Inf. Softw. Technol.*, 47(8):521–532, Jun 2005.

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

[20] 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.

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

[22] D. Klein and D. Manning, Christopher. Accurate unlexicalized parsing. In *Proc. 41st Meeting of the Association for Computational Linguistics*, pages 423 – 430, 2003.

[23] 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.

[24] H. Lee, Y. Peirsman, A. Chang, N. Chambers, M. Surdeanu, and D. Jurafsky. Stanford's multi-pass sieve coreference resolution system. In *Proc. CoNLL-2011 Shared Task*, 2011.

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

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

[27] M. Mossienko. Automated Cobol to Java recycling. In *Proc. 7th CSMR*, pages 40–, 2003.

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

[29] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: towards automating risk assessment of mobile applications. In *Proc. 22nd USENIX Security Symposium*, 2012.

[30] 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.

[31] 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.

[32] K. Raghunathan, H. Lee, S. Rangarajan, N. Chambers, M. Surdeanu, D. Jurafsky, and C. D. Manning. A multi-pass sieve for coreference resolution. In *Proc. EMNLP*, 2010.

[33] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Trans. on Software Engineering*, 39(5):613–637, 2013.

[34] 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.

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

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

[37] M. Stickel and M. Tyson. *FASTUS: A Cascaded Finite-state Transducer for Extracting Information from Natural-language Text.* MIT Press, 1997.

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

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

[40] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. 21st ICSE*, pages 246–255, 1999.

[41] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. on Software Engineering*, 14(8):1207–1228, 1988.

[42] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proc. 33rd ICSE*, pages 474–484, 2011.

[43] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *Proc. 20th FSE*, 2012.

[44] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *Proc. 20th ACM SIGSOFT FSE*, pages 12:1–12:11, 2012.

[45] W. Zheng, Q. Zhang, and M. Lyu. Cross-library API recommendation using web search engines. In *Proc. 13th ESEC/FSE*, pages 480–483, 2011.

[46] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. 32nd ICSE*, pages 195–204, 2010.

[47] 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.

[48] 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.