Inferring Method Specifications from Natural Language API Descriptions

Rahul Pandita*, Xusheng Xiao*, Hao Zhong†, Tao Xie*, Stephen Oney‡, and Amit Paradkar§
*Department of Computer Science, North Carolina State University, Raleigh, USA
†Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing, China
‡Human-Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, USA
§I.B.M. T. J. Watson Research Center, Hawthorne, NY, USA
{rpandit, xxiao2, txie}@ncsu.edu, zhonghao@itechs.iscas.ac.cn, soney@cs.cmu.edu, paradkar@us.ibm.com

***Abstract*—Application Programming Interface (API) documents are a typical way of describing legal usage of reusable software libraries, thus facilitating software reuse. However, even with such documents, developers often overlook some documents and build software systems that are inconsistent with the legal usage of those libraries. Existing software verification tools require formal specifications (such as code contracts), and therefore cannot directly verify the legal usage described in natural language text in API documents against code using that library. However, in practice, most libraries do not come with formal specifications, thus hindering tool-based verification. To address this issue, we propose a novel approach to infer formal specifications from natural language text of API documents. Our evaluation results show that our approach achieves an average of 92% precision and 93% recall in identifying sentences that describe code contracts from more than 2500 sentences of API documents. Furthermore, our results show that our approach has an average 83% accuracy in inferring specifications from over 1600 sentences describing code contracts.**

I. Introduction

Software reuse is commonly practised since the advent of software development [9]. Software reuse facilitates development of larger, more complex, and timely-delivered software systems [10]. To determine what and how to reuse, specifications of reusable software libraries play an important role. In the absence of specifications, developers may write code that is inconsistent with the expectations of libraries. As a result, not only such code is of inferior quality and contains faults, the added cost of debugging and correcting such faulty code could also defeat the purpose of reusing software.

Code contracts [4], [20] have emerged as a popular way of formalizing method specifications close to the implementation level. Code contracts unambiguously capture the expectations of a method in terms of what is required (*preconditions*) and what to expect after method execution (*postconditions*). Furthermore, code contracts can be subjected to formal verification by existing state-of-the-art verification tools such as Spec# [1], JML[1], and Code Contracts for .NET[2]. Additionally, code contracts can be used for formal proofs and automated code correction [38].

Despite being highly desirable, code contracts do not exist in a formalized form in most existing software systems in practice [24]. In contrast, library developers commonly describe legal usage in natural language text in Application Programming Interface (API) documents. Typically, such documents are provided to client-code developers through online access, or are shipped with the API code. For example, J2EE's API documentation[3] is one of the most popular API documents.

Even with such documents, client-code developers often overlook some API documents and use methods in API libraries incorrectly [22]. Since these documents are written in natural language, existing tools cannot verify legal usage described in a library's API documents against the client code of that library. One possible solution is to manually write code contracts based on the specifications described in API documents. However, due to a large number of sentences in API documents, manually hunting for contract sentences and writing code contracts for the API library is prohibitively time consuming and labor intensive. For instance, the `File` class of the C# .NET Framework has around 800 sentences. Moreover, not all of these sentences describe code contracts, requiring extra effort to first locate the sentences describing code contracts and then translate them.

To address the preceding problem, we propose a novel approach to facilitate verification of legal usage described in natural language text of API documents against client code of those libraries. We propose new techniques that apply Natural Language Processing (NLP) on method descriptions in API documents to automatically infer specifications. In particular, our techniques address the following challenges to infer specifications automatically.

Ambiguity. Existing linguistic analysis techniques focus on well-written documents such as news articles [30]. In contrast, method descriptions are often not well-written. For instance, consider the sentence from the `File` class in the C# .NET Framework: *"true if path is an absolute path; otherwise false"*. This sentence does not have the main subject and the verb.

To address this challenge, we use meta-data such as position information of description as well as method signatures. In particular, in this example, the sentence is placed under the return descriptions and the method signature describes the return type as boolean. We use this information to infer that words "true" and "false" describe the return value of the method.

[1]http://www.eecs.ucf.edu/~leavens/JML/
[2]http://research.microsoft.com/en-us/projects/contracts/
[3]http://download.oracle.com/javaee/1.6/api/

Programming Keywords. Method descriptions often contain programming keywords (e.g., `true, null, buffer`), which have a different meaning in the context of programs, in contrast to general linguistics. For instance, consider this sentence from the `Path` class in the C# .NET Framework: *"This method also returns false if path is null"*. In this sentence, words 'false' and 'null' are nouns in the context of object-oriented languages such as Java and C#, whereas in general linguistics these words are adjectives. Thus, these keywords need to be handled differently.

For those keywords, we propose a new technique called *Noun Boosting* to distinguish keywords from the other words.

Semantic Equivalence. A legal usage in natural language can be described in different words and semantic structures. For instance, consider the following two fragments that describe the same specification: *"name can contain numbers, underscores..."* and *"name consists of numbers and/or underscores..."*. Thus, there is a need to identify the semantic equivalence of legal usage described in different ways.

To address this challenge, we propose a new technique called *equivalence analysis* based on identified grammatical structures (main nouns and verbs) of a sentence.

In summary, our approach infers specifications from existing API documents, thus facilitating verification of client code of an API library against the natural language descriptions of the library. As our approach analyzes API documents in natural language, it can be reused independent of the programming language of the library. Additionally, our approach complements existing approaches [5], [21], [37] that infer code contracts from source code or binaries. To the best of our knowledge, ours is the first approach that analyzes API documents to extract specifications targeted towards generating code contracts.

Our paper makes the following major contributions:

- A technique that effectively identifies natural language sentences (in the API documents for a library) that describe code contracts, hereby referred to as contract sentences, and a technique to infer specifications from the identified contract sentences.
- A prototype implementation of our approach based on extending the Stanford Parser [17], [30], which is a natural language parser to derive the grammatical structure of sentences. An open source implementation of our prototype can be found at our website[4].
- An evaluation of our approach on 2717 sentences (for 333 methods) in five different classes from the .NET Framework and Facebook API for C#. Our evaluation results show that our approach effectively identifies contract sentences with an average of 92% precision and 93% recall. Additionally, our approach infers specifications from around 1700 contract sentences with an average accuracy of 83%.

[4]http://research.csc.ncsu.edu/ase/projects/pint/

II. Background

A. Code Contracts

Code contracts, based on the Design by Contracts (DbC) [4] methodology, are typically in the form of method pre-conditions, post-conditions, and class invariants. They are used to specify what a method accomplishes without giving details of how the method is implemented. Pre-conditions for a method describe what is expected by the method in terms of inputs. Post-conditions for a method describe what to expect when the method has finished execution in terms of output. Class invariants describe what conditions on receiver objects of the class are true before and after the execution of each method in the interface of the class. Furthermore, code contracts are useful for software reuse, software testing, formal proofs, and automated correction of code [38].

B. NLP Preliminaries

Due to the inherent "ambiguous" nature of natural language, it is very difficult to "understand" and convert the natural language text into precise and unambiguous specifications that can be processed by computers. In contrast, existing NLP techniques have been shown to be fairly accurate in highlighting the grammatical structure of a natural language sentence [17], [19] . We next present two of the NLP techniques that have been used in this work.

Word Tagging/Parts Of Speech (POS) tagging [17] aims to identify the part of speech (such as nouns and verbs) that a particular word within a sentence belongs to. The most commonly known technique is to train a classification parser over a manually labelled dataset [36]. Current state-of-the-art approaches can achieve around 97% accuracy over well written news articles [30].

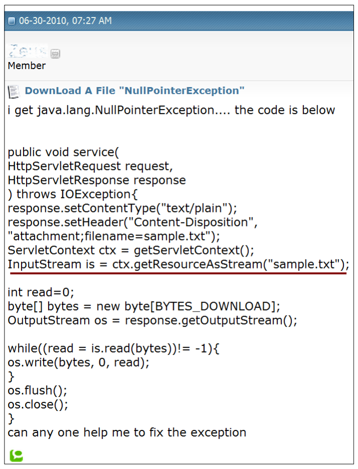
Phrase and clause parsing (Chunking) [17] divides a sentence into a set of words that are logically related such as a *Noun Phrase* and *Verb Phrase*. Chunking thus further brings forth the syntax of a sentence after POS tagging has been done. Current state-of-the-art approaches can achieve around 90% accuracy in classifying phrases and clauses [30] over well written news articles.

III. Motivating Examples

We next present two examples where developers wrote faulty code because they overlooked legal usage in method descriptions. We collected these examples from online developer forums. These examples highlight the importance of our approach, since the defects shown in them could have been easy to fix with our inferred code contracts.

NullPointerException. As shown in Figure 1, a developer asked a question (on `06-30-2010`) in one of the Java forums[5] regarding `java.lang.NullPointerException`.

[5]http://www.java-forums.org/java-servlet/30289-download-file-nullpointerexception.html



06-30-2010, 07:27 AM

Zeus
Member

DownLoad A File "NullPointerException"

i get java.lang.NullPointerException.... the code is below

```
public void service(
HttpServletRequest request,
HttpServletResponse response
) throws IOException{
response.setContentType("text/plain");
response.setHeader("Content-Disposition",
"attachment;filename=sample.txt");
ServletContext ctx = getServletContext();
InputStream is = ctx.getResourceAsStream("sample.txt");

int read=0;
byte[] bytes = new byte[BYTES_DOWNLOAD];
OutputStream os = response.getOutputStream();

while((read = is.read(bytes))!= -1){
os.write(bytes, 0, read);
}
os.flush();
os.close();
}
```

can any one help me to fix the exception

Figure 1. Description of a question posed in a Java forum

Figure 1 shows the code snippet. After going through the suggestions from the contributing forum members, the author of this question was finally able to resolve the problem to a path issue occurring in the underlined line, a month (`07-30-2010`) after he initially posted the question.

The associated API documents reveal that there are many ways to throw `java.lang.NullPointerException`, thus making the task of debugging non-trivial. The problem caused due to invoking the `read` method on a `null` object of `InputStream`. The method description for the `getResourceAsStream` method in the `ServletContext` interface states that *"This method returns null if no resource exists at the specified path."*. The sentence, once translated into a formal specification, can be verified statically, or at runtime to pinpoint the problem. For instance, given the specification that the method can return `null`, advanced Integrated Development Environments (IDEs) can raise a warning after the developer attempts to perform an operation on the return value of the `getResourceAsStream` method, thus asking the developer to put the necessary checks in place.

Path format not supported. Another developer posted a 44-line code snippet in a forum[6]. The developer reported that he/she was noticing exceptions while moving a file. After a brief exchange of messages, the issue was traced to the `moveTo` method in the `FileInfo` class in C# .NET. Further exchanges revealed that the developer was attempting to move and rename the file to "11-19-2009_5:48:27.txt". The method description for the `moveTo` method in `FileInfo` states that the method throws an `ArgumentException` if the new file name consists of invalid characters defined in `Path.getInvalidCharacters()` (including':' character). After our approach infers a code contract from the description, the problem can be identified and reported by static checkers that are usually built into IDEs.

[6]http://www.dreamincode.net/forums/topic/140470-elusive-error-while-renaming-file/

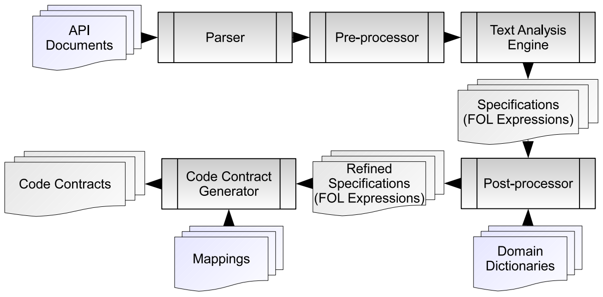


Figure 2. Overview of our approach

IV. Approach

We next present our approach for inferring code contracts from method descriptions. Figure 2 gives an overview of our approach. Our approach uses a parser, a pre-processor, a text analysis engine, a post-processor, and a code contract generator. The parser accepts the API documents and extracts intermediate contents from the method descriptions. The pre-processor augments the sentences in an intermediate representation with meta-data. The text analysis engine accepts the intermediate representation of the sentences, and then based on our semantic templates, generates specifications in the form of First-Order Logic (FOL) expressions. The post-processor refines the FOL expressions. The code contract generator accepts the FOL expressions and generates code contracts by using a mapping relation to the constructs of the target programming language.

A. Parser

Our parser accepts API documents and extracts intermediate contents from the method descriptions. In particular, from the method descriptions, our parser extracts the following contents: (1) *summary description*: the summary of the method; (2) *argument description*: the descriptions of the method's arguments; (3) *return description*: the descriptions of the method's return value; (4) *exception description*: the descriptions of exceptions explicitly thrown by the method; (5) *remark description*: additional descriptions about the functionality of the method.

B. Pre-processor

Our pre-processor accepts extracted contents of the method descriptions, and performs three major tasks.

Meta-data augmentation. For each identified method, our pre-processor collects the following meta-data information and associates it with respective sentences: (1) the names and data types of method arguments; (2) the types of the return value and exceptions; (3) the names of the classes, namespaces, and methods. For example, for the method description shown in Figure 4, the meta-data information associated with the sentences in Line 03 is as follows: (1) Sentence Type: Argument Description; (2) Argument Name: `prop_name`; (3) Argument Type: `String`.

This information is used in code contract generation by substituting the name of the variable with its place-holder and matching a template for code contracts using the data type of the variable. In particular, the pre-processor uses method signatures and their associated tags for meta-data augmentation. From the method signatures, our approach extracts the name of the method arguments, the data types of the method arguments, and the exceptions thrown by the method.

Noun Boosting. Since our text analysis engine uses the Parts-Of-Speech (POS) tags provided by a POS-tagger, the accuracy of the inferred specifications is dependent on the accuracy of the POS-tagger. However, there are specific words that represent nouns in the context of programs, in contrast to adjectives or verbs in the context of general linguistics. For example, consider the statement *"This method also returns false if path is null"*. In this sentence, "false" and "null" should be treated as nouns since they are constructs of programming languages, but a typical POS tagger would incorrectly classify them as adjectives.

Our pre-processor identifies these words from the sentences based on a domain specific dictionary, and thus forces the underlying POS tagger to identify them as nouns. In particular, our pre-processor uses a predefined list of words for noun boosting. We manually collected these words by looking into the method descriptions in the `Data` class of the Facebook API and the `Path` class of the .NET Framework API. A list of these words is available on our project website.

Programming Constructs and Jargon Handling. In English grammar, the "." character represents the end of a sentence. However, in programming languages, the "." character is used as a separator character as well. For example, in the `Facebook.Data` namespace, the "." character represents that the `Facebook.Data` namespace exists within the `Facebook` namespace. Our pre-processor identifies these separators, and replaces "." with "_". For example, `"Facebook.Data"` is replaced with `"Facebook_Data"`.

Additionally, developers tend to use abbreviations for specific words (e.g., max. for maximum and min. for minimum). Our pre-processor identifies these words, and replaces abbreviations with their full names. For example, "max." is replaced with "maximum".

These techniques increase the accuracy of the underlying POS tagger, and thus increase the accuracy of our text analysis engine. Furthermore, our pre-processor maintains mapping relations of the place-holder words from the programming language constructs to the original words and locations, and these relations are used by our post-processor later to infer specifications.

Methods and namespaces have a well-defined lexical structure in a programming language. Our pre-processor uses this structural information, and builds regular expressions to identify these words. For handling jargons and abbreviations such as "max.", we manually built a list of such words. In future work, we plan to adapt Hill et al.'s technique [16] to generate the list automatically.

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. Text Analysis Engine

Our text analysis engine parses pre-processed sentences, and builds specifications in the form of FOL expressions. We chose FOL, since previous research [28], [29] shows that FOL is an adequate representation for natural language analysis.

We first use a POS tagger to annotate POS tags in a sentence. We then use an NLP technique, called shallow parsing [2]. A shallow parser accepts the lexical tokens generated by the POS tagger and attempts to classify sentences based on pre-defined semantic templates. Shallow parsing is implemented as a sequence of cascading finite state machines. Research [2], [13], [28], [32] has shown the effectiveness of using finite state machines in different areas of linguistic analysis such as morphological lookup, POS tagging, phrase parsing, and lexical lookup.

Table I shows frequently used semantic templates for identification of specifications. Column "Description" describes what is inferred from the sentence if a semantic pattern holds. For example, for the template described in the first row in Table I, the FOL expression is constructed as ***can not be*** *(path, null)*, where "path" and "null" are terms to the predicate "can not be". The specification is interpreted as: "can not be" predicate should be evaluated to be true over terms "path" and "null".

As another example, our text analysis engine uses the semantic pattern, *transitive predicate*, described in the fourth row in Table I to analyze the sentence in Line 3 of Figure 4. Figure 3 shows the graphical FOL expression. Each internal node (shaded grey) represents a predicate and the children of these nodes represent the terms to that predicate.

We implemented a configurable infrastructure to accept a POS tagger to annotate a sentence with POS tags. In particular, for our evaluation, we used the Stanford Parser [17], which is a natural language parser to work out the grammatical structure of sentences. The Stanford Parser parses a natural language sentence and determines POS tags associated with different words/phrases. We also implemented a generic and extensible framework that accepts semantic patterns based on the functions of POS tags and converts them into a series of cascading FSMs. Once POS tags have been determined by a POS tagger, the sentences along with tags are passed as an input to the shallow parser, which generate FOL expressions based on the FSMs.

D. Post-processor

Our post-processor accepts the FOL expressions produced by the previous component and performs three types of semantic analysis: removing irrelevant modifiers in predicates, classifying predicates into a semantic class based on domain dictionaries, and augmenting expressions.

Table I
CATEGORIES OF SHALLOW PARSING SEMANTIC TEMPLATES.

	Name	Example	Description
1.	Predicate (Name)	The (path)$_{subject}$ (can not be)$_{verb}$ null$_{object}$	The subject and object form the terms of the predicate represented by verb.
2.	Conditional followed or preceded nominal predicate	If (path does not have extension)$_{conditional}$, (GetExtension)$_{subject}$ (returns)$_{verb}$ (System.String.Empty)$_{object}$	The subject-verb-object forms specification as described in row 1, which is true when the condition highlighted by $conditional$ is true. The condition is further resolved using one of the templates.
3.	Prepositional predicate	(Path)$_{subject}$ (is)$_{verb}$ (not null or empty String)$_{preposition}$	The verb forms the partial predicate and the subject forms one of the terms. The second term and the remaining of the predicate are extracted by resolving the preposition.
4.	Transitive predicate	(Name)$_{subject}$ (is)$_{verb}$ a (valid , identifier)$_{object-subject}$, which (is no longer than 32 characters)$_{clause}$	The sentence is broken down into two sentences. The first sentence ends with the phrase labeled $_{object-subject}$, and the second sentence begins with the phrase labeled $_{object-subject}$. Each sentence is further resolved and the resulting specifications are joined using the logical AND operator.

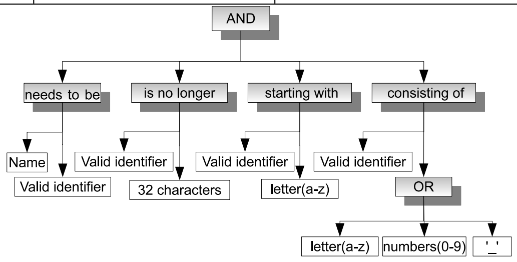


Figure 3. Specifications in format of FOL expressions extracted by our NLP Parser for `DefineObjectProperty` method in Facebook API

```
01:/// <summary>
02: .....
03:/// <param name=``prop_name''> This name
    needs to be a valid identifier, which
    is no longer than 32 characters, starting
    with a letter (a-z) and consisting of only
    small letters (a-z) numbers (0-9), and/or
    underscores.</param>
04: .....
05:public void DefineObjectProperty(string
    obj_type, string prop_name,
        int prop_type)
```

Figure 4. The method description of the `DefineObjectProperty` method in Facebook API

Equivalence analysis. Consider the predicate, *'needs to be'*, in FOL representation shown in Figure 3. The words, *"needs to"*, are modal modifiers to the verb *'be'*. Such modal modifiers are identified and eliminated. Furthermore, our post-processor classifies predicates into pre-defined semantic classes based on domain dictionaries. This classification addresses the challenge of inferring semantic equivalence. For instance, the predicate, "starting with", in Figure 3 can also be represented as "begins with". Our post-processor identifies and classifies all semantically equivalent predicates into a single category, and thus reduces the effort to individually write mappings for every predicate in inferred FOL expressions even when they represent same the semantic function. We have identified the following seven major semantic categories for predicates: (1) Greater, (2) Lesser, (3) Begin, (4) End, (5) Consist, (6) Equal, and (7) Action with respect to expressions dealing with code contracts. The negative semantic categories are represented using a negation operator preceding the identified semantic class.

In the preceding semantic analysis, our post-processor uses an NLP technique called lemmatization [30]. Lemmatization involves full morphological analysis to accurately identify the lemma for each word. Extracting lemmas reduces the various operational forms of a word to its root. For example, "am", "are", and "is" are all reduced to "be". Once the lemma of a word is identified, our post-processor uses the lemma to query a synonym from the WordNet [7] database for a suitable replacement. From the implementation perspective, we maintain a list of modifier words to identify and discard them. We have also collected synonyms from WordNet to classify a predicate in one of the semantic classes. If a match is not found, our post-processor places the predicate in the unknown category.

Intermediate Term Elimination. The intermediate term elimination attempts to remove intermediate terms, if they are found in the extracted expressions. For example, consider the statement in Line 3 of Figure 4. Here, `Valid Identifier` is used as the intermediate term to establish the `"no longer"` relationship between `"name"` and `"32 characters"`. Since a shallow parser is independent of the semantics of the words used in a sentence, our text analysis engine picks up these intermediate terms as valid arguments to the predicate using them, as shown in Figure 3. These terms are of no inherent importance in code contract generation.

Our post-processor identifies such terms and eliminates them by replacing their usage with their definition. In particular, our post-processor eliminates intermediate terms by parsing FOL expressions. We specifically watch out for terms that are involved in an equality operator with a variable name followed by the same term being used as an input to another predicate in the representation.

Expression Augmentation. The sentences in return descriptions and exception descriptions in an API document are

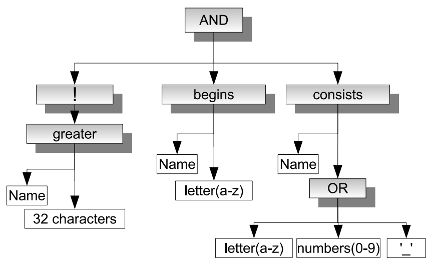


Figure 5. FOL expression after synonym analysis and compaction for the `DefineObjectProperty` method in Facebook API

Algorithm 1 Expression Augmentation generator

```
Input: Expr $e$, Meta-data $d$
Output: Expr $e'$
 1: $Expr\ e' = e$
 2: if $(d.description == return)$ then
 3:     if $(e'.root == \text{“} \to \text{”})\&\&(e'.right\ is\ variable)$ then
 4:         $Term\ t = e'.right$
 5:         if $findType(t) == d.returnType$ then
 6:             $Predicate\ p = new\ Predicate(\text{“}returns\text{”})$
 7:             $p.term = t$
 8:             $e'.right = p$
 9:         end if
10:     end if
11: end if
12: if $(d.description == exception)$ then
13:     if $(e'.root == \text{“} \to \text{”})\&\&(e'.right\ is\ empty)$ then
14:         $Term\ t = d.exception\_name$
15:         $Predicate\ p = new\ Predicate\ (\text{“}throw\text{”})$
16:         $p.term = t$
17:         $e'.right = p$
18:     end if
19:     if $(e'.root! == \text{“} \to \text{”})$ then
20:         $Term\ t = d.exception\_name$
21:         $Predicate\ p = new\ Predicate\ (\text{“}throw\text{”})$
22:         $p.term = t$
23:         $Expr\ e'' = new\ Expr(\text{“} \to \text{”})$
24:         $e''.left = e'$
25:         $e''.right = p$
26:         $e' = e''$
27:     end if
28: end if
29: return $e'$
```

often not well written. For example, consider the following sentences:

1) *“true if path is an absolute path; otherwise false.”*— the return descriptions for the`IsPathRooted` method in the `Path` class in the C# .NET Framework. The main subject and verb are missing as in what is true and false.
2) *“If path is null.”*— one of the exception descriptions repeated in many methods in the `File` class in the C# .NET Framework. The action is missing as in what happens if the path is null.
3) *“IO error occurs while accessing specified directory.”*— one of the exception descriptions repeated in many methods in the `Directory` class in the C# .NET framework. While the sentence describes a code contract, the sentence omits important information in terms under what specific condition the exception is thrown.

Our expression augmentation attempts to augment these expressions. In particular, for each method, we use meta-data collected in the pre-processor augment to complete the FOL expressions involving return and exception descriptions. Here, we propose Algorithm 1 to achieve our expression augmentation. The algorithm accepts an FOL expression and the meta-data of a sentence. The algorithm returns an augmented expression if successful, and otherwise returns the original expression. The algorithm first checks whether the expression corresponds to a return description statement (Line 2). If the expression is a conditional expression and the right hand side of the expression is a variable term, the algorithm checks whether the type of the variable matches the return type described in the meta-data. Literals, ‘true’, and ‘false’, are identified as boolean; ‘numeric values’ are identified as numeric that matches integer, float, and double. If a match is found, we construct the right hand side of the original predicate as **returns**.

For the descriptions of exceptions, our expression augmentation does a similar check except that there is no need to match the type of a variable term. We construct the predicate as **throws**. Additionally, for the expressions in exception descriptions where no conditional expression is identified, we explicitly construct a conditional FOL expression (Line 23-25) and associate the expression to the left hand side and the throws predicate to the right hand side.

E. Code Contract Generator

Our code-contract generator generates code contracts from the extracted FOL expressions. The generator uses the predefined mapping of semantic classes of the predicates to the programming constructs to produce valid code contracts. Our current implementation supports the mapping relations for the `String` class, `Integer` class, `null` checks, `return` and `throws` constructs. With more mapping relations, our generator can easily produce code contracts involving complex objects.

For example, consider the FOL expression in Figure 5. The “greater” predicate is mapped to the `length` method of the `String` class. Thus, the resulting code contract is `requires(!(name.length()>32))`. In contrast, “begins” is mapped to the `startswith` and `substring(0,1)` methods of the `String` class. Our generator resolves which methods to choose by taking into account the argument for the method. If the argument is a character (characterized by a single character in quotes) or string (characterized by a string in quotes), our generator uses the `startswith` method, and if the argument is a range (characterized by expression ‘a–z’), our generator uses the `substring(0,1)` method by converting the range to a regular expression. Thus, the final contract is `requires(name.substring(0, 1).matches("[a-z]+"))`.

```
01:requires(!prop_name.length()>32)
02:requires(prop_name.substring(0,1).
   matches([a-z]+))
03:requires(prop_name.matches(([a-z][0-9][_])*))
```

Figure 6. The inferred specifications for the `prop_name` variable of the `DefineObjectProperty` method in Facebook API

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 our approach in identifying contract sentences (i.e., sentences that describe code contracts)?
- **RQ2**: What is the accuracy of our approach in inferring specifications from contract sentences in the API documents?
- **RQ3**: How do the specifications inferred by our approach compare with the human written code contracts?

A. Subjects

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

C# File System API documents. These documents describe correct usage of methods for manipulating files in the .NET environment. However, developers still post a lot of questions regarding their usage. Because of the importance of the File API, we chose these API documents as the first set of subject documents for inferring specifications. In particular, we use three key classes (`File`, `Path`, and `Directory`) in our evaluations.

Facebook API documents. Facebook is a popular social networking site, which allows developers to write their own third-party applications. According to Facebook statistics, people on Facebook install 20 million applications everyday[7]. Due to the sheer popularity of Facebook and a huge number of developers developing third-party applications, we chose the Facebook API [8]for C# as another set of subject documents for our evaluation. In particular, we use four key classes (`Data`, `Friends`, `Events`, and `Comments`) within the Facebook API for our evaluations.

Table II shows the statistics of the subject documents used in our evaluations. Column "Class[API Library]" lists the name of classes and their corresponding libraries. Column "$\#M$" lists the number of methods in each class. Column "$\#S$" lists the number of natural language sentences in method descriptions of each class.

B. Evaluation Results

1) RQ1: Precision and Recall in Identifying Contract Sentences: In this section, we quantify the effectiveness of our approach in identifying contract sentences by answering RQ1. We first manually measured the number of contract sentences in the API documents. We considered a sentence as a contract sentence if it contains a clause that is either a pre-condition or post-condition. Two authors independently labeled sentences as contract sentences by discussing iteratively until they reached a consensus. We then applied our approach on the API documents and manually measured the number of `true positives` (TP), `false positives` (FP), and `false negatives` (FN) produced by our approach as follows:

- **TP**. A sentence that is a contract sentence and is identified by our approach as a contract sentence.
- **FP**. A sentence that is not a contract sentence and is identified by our approach as a contract sentence.
- **FN**. A sentence that is a contract sentence and is identified by our approach as not a contract sentence.

In statistical classification [23], Precision is defined as the ratio of the number of true positives to the total number of items reported to be true, and Recall is defined as the ratio of the 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 values of Precision, Recall, and F-Score indicate higher quality of the contract statements inferred using our approach. Based on the total number of TP, FP, and FN, we calculated the *Precision*, *Recall*, and *F-score* of our approach in identifying contract sentences as follows:

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

Table II shows the effectiveness of our approach in identifying contract sentences. Column "Class[API Library]" lists the names of the classes. Column "#S" lists the number of sentences in each class, and Column "S_C" lists the number of sentences manually identified as contract sentences. Columns "TP", "FP", and "FN" represent the number of `true positives`, `false positives`, and `false negatives`, respectively. Columns "P", "R", and "F_S" list values of `precision`, `recall`, and `f-scores`, respectively. Our results show that, out of 2717 sentences, our approach effectively identifies contract sentences based on average Precision, Recall and F-score of 91.8%, 93% and 92.4% respectively.

We next present an illustrative example of how our approach incorrectly identifies a sentence as a contract sentence. Consider the sentence from the `getLastWriteTime` method description in the `Directory` API for C#: *"The file or directory for which to obtain write date and time information."* The sentence describes the input parameter `path`. Ideally, a POS tagger should parse the statement as a noun-phrase statement, i.e., a sentence including just a noun-phrase. However, the POS tagger incorrectly annotates this sentence as including a subject, object, and predicate, where the predicate is "write". Since our shallow parser is dependent on the POS tagger to correctly annotate POS tags, our approach incorrectly identifies this sentence as a contract sentence. The FP produced by our approach are primarily due to the incorrect POS tags annotated by the POS tagger.

[7]https://www.facebook.com/press/info.php?statistics

[8]http://facebooktoolkit.codeplex.com.

Table II
STATISTICS OF SUBJECT CLASSES AND EVALUATION RESULTS

Class [API Library]	$\#M$	$\#S$	S_C	TP	FP	FN	P	R	F_S	S_I	Acc	S_D	C	Q
Data[Facebook.Rest]	133	810	320	288	55	32	84.0	90.0	86.9	244	76.3	102	21	0.75
Friends[Facebook.Rest]	37	215	126	96	10	30	90.6	76.3	82.8	84	66.7	17	0	0.83
Events[Facebook.Rest]	29	194	122	110	12	12	90.2	90.2	90.2	84	68.9	15	0	0.85
Comments[Facebook.Rest]	16	96	33	33	19	0	63.5	100.0	77.7	28	84.9	12	0	0.70
File[System.IO(.NET)]	56	795	647	627	15	20	97.7	97.0	97.3	599	92.6	NA	NA	NA
Path[System.IO(.NET)]	18	99	63	48	11	15	81.4	76.2	78.7	44	69.8	NA	NA	NA
Directory[System.IO(.NET)]	44	508	380	371	18	9	95.4	97.6	96.5	327	86.1	NA	NA	NA
Total	333	2717	1691	1573	140	118	91.8*	93.0*	92.4*	1410	83.4*	146	21	0.79*

* Column average

The FN in our approach are also primarily due to incorrect POS tags annotated by the POS tagger. Overall, a significant number of FP and FN can be further reduced by improving the existing underlying NLP infrastructure.

2) RQ2: Accuracy in Inferring Specifications from Contract Sentences: To address RQ2, we apply our approach on sentences that were manually identified as contract sentences to infer FOL expressions. We then manually verify the correctness of the inferred specifications. We define the `accuracy` of our approach as the ratio of the contract sentences with correctly inferred expressions to the total number of contract sentences.

Table II shows the effectiveness of our approach in inferring specifications (FOL expressions) from contract sentences. Column "Class[API Library]" lists the name of the classes. Column "S_C" lists the number of sentences manually identified as contract sentences. Column "S_I" lists the number of specifications that were correctly inferred from contract sentences. Column "Acc" lists the accuracy of our approach in inferring specifications from the contract sentences. Our results show that, out of 1691 contract sentences, our approach correctly inferred specifications from 1410 contract sentences, with the accuracy of 83.4%.

We next present an illustrative example of how our approach infers an incorrect specification from a contract sentence. Consider the sentence from the `Friends` class in the Facebook API for .NET: *"The first array specifies one half of each pair, the second array the other half; therefore, they must be of equal size."*. The sentence describes the two input parameters. Our approach successfully identifies the sentence as a contract sentence. However, while inferring the specification, our approach faces difficulty in accurately inferring the semantic relations. In particular, the complexity of the sentence (involving both code contracts and generic descriptions) makes it difficult for the POS tagger to correctly annotate POS tags, thus causing a semantic pattern to be incorrectly applied to the sentence. This sentence appears 20 times across different method descriptions in the `Friends` class where our approach performed the worst. If our approach would have correctly inferred specifications from the sentence, the accuracy of our approach for the Friends API would have been 82.5% instead of 66.7%.

3) RQ3: Comparison with Human Written Contracts: To answer RQ3, we compared the specifications inferred from contract sentences by our approach with the human written code contracts. The Facebook API for C# is equipped with code contracts that were written by Rubinger et al. [26] as a part of their experience report on applying the Microsoft Code Contract system for the .NET framework. We first manually calculated S_I as the number of specifications correctly inferred by our approach for a class. We then calculated S_D as the number of code contracts written by Rubinger et al. for that class, and C as the number of specifications in common.

Table II shows the comparison of the specifications inferred by our approach to the human written contracts. Column "Class[API Library]" lists the name of the classes. Column "S_I" lists the number of specifications correctly inferred by our approach. Column "S_D" lists the number of human written code contracts. Column "C" lists the number of the specifications that are common between "S_I" and "S'_D". Our results show that out of 440 inferred specifications and 146 human written contracts only 21 are in common. We next discuss some of the implications of the results.

Before carrying out this evaluation, we had hoped that the specifications inferred by our approach would largely be a superset of the human written contracts, as Rubinger et al. claimed to have written these contracts as a *"direct translation of the method descriptions and some as their own interpretation of the API"* [26]. However, the results suggest that not to be the case. We were intrigued by the outcome and manually investigated the nature of the human written contracts and the specifications inferred by our approach. Interestingly, we found that all of the human written code contracts are assertions categorized as follows: (1) Null Checks, (2) Range Checks, and (3) Size Checks. Furthermore, around 80% of these are simplistic `not null` and `length>0` checks. For instance, for the example method description in Figure 4, the human written code contracts are:

```
Contract.Requires(name!=null&&name.Length>0);
```

Although the contract holds true, it is a simplistic `not null` and `length>0` check, since it fails to capture limitations of the first character and size restrictions (`<32`). In contrast, our approach is capable of inferring detailed specifications as shown in Figure 6. Furthermore, there is no direct text (in the method description) that corresponds to some of human written contracts. Since our approach infers specifications from only method descriptions, we do not produce such specifications for these contracts. Additionally, of the 22 instances of the same description as shown in Figure 4 for input parameter `name` across the methods in the `Data` class for the Facebook API, we found only 2 (9%) instances where the specifications inferred by our approach completely matched the code contracts written by Rubinger et al. The remaining 20 (91%) instances were translated as described earlier. Logically, specifications produced by our approach imply the corresponding human written code contracts. In future work, we plan to explore techniques to infer these implied contracts. On manual examination of other human written contracts, we concluded that, despite valid, several contracts are deemed to be implied and hence there is no corresponding textual description in the API method descriptions, including all the contracts in the `Friends`, `Comments`, and `Events` classes and more than 70% of the contracts in the `Data` class of the Facebook API. These contracts are the ones that Rubinger et al. claimed to have written as their own understanding of the API. In addition, none of the human written contracts capture post-conditions. In contrast, our approach is able to infer both pre- and post-conditions from the method descriptions.

From our evaluation, we conclude that our approach systematically infers specifications from the method descriptions in API documents. However, there are cases where method descriptions do not completely describe specifications. In such cases, our approach can work in conjunction with either human written contracts or approaches that statically or dynamically infer code contracts from API implementation [5], [21].

4) Summary: In summary, our evaluation shows that our approach effectively identifies contract sentences from the method descriptions in API documents, demonstrated by the high values in Columns "Precision", "Recall", and "F-score" in Table II from over 2500 sentences. Our evaluation also shows that our approach infers specifications from the contract sentences with high accuracy (averagely 83.4%), as shown by the values in column "Accu" in Table II from over 1600 contract sentences. Furthermore, our evaluation results show that our approach can infer detailed specifications than human written contracts.

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 projects: one commercial and the other open source. The C# File System API documents describe one of the most commonly used and mature APIs. We also used the Facebook API for C#, which is relatively new (introduced in 2008). 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. Additionally, to represent human written code contracts, we used the human written code contracts for the Facebook API for C# [26], and did not use any code contracts written by ourselves. Threats to internal validity include the correctness of our implementation in extracting code contracts and labelling a statement as a contract statement. To reduce the threat, we manually inspected all the specifications 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 specifications in the natural language texts of 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.

Code Searching. Code searching [25], [34] for reuse is a classic problem [9] in software engineering. Among previous approaches, a recent approach by Riess [25] provides promising results by using semantics such as code contracts as input-output relationships for code searching. Our approach can be used for generating specifications from API documents in a code repository and thus assisting such approaches in producing better results.

Program Synthesis. Automated program synthesis holds potential for easing the task of a developer by taking care of program generation and allowing the developer to concentrate on design tasks. Recent work by Srivastava et al. [31] addresses the problem by leveraging specifications in the form of pre/post-conditions and invariants to achieve synthesis. Our approach can work in conjunction with such approaches to extract specifications from natural language text to achieve better synthesis.

We next present some of the limitations of our approach.

Information flow analysis. Our approach currently takes into account the specifications described in a single sentence. However, there are instances when a specification is distributed across several sentences. Consider the sentences below:

"parameter values:Id-value pairs of preferences to set. Each id is an integer between 0 and 200 inclusively. Each value is a string with maximum length of 128 characters."

The first sentence describes the data structure used for the variable values. The sentences following the first sentence describe the specification on each item in the data structure. Since currently our approach works on individual sentences, it is not possible to establish the relationship between the specifications described in later sentences to the first sentence. In future work, we plan to investigate techniques to facilitate information flow analysis to handle such situations.

Contextual Information. Some API documents are not comprehensive. Method descriptions omit certain specifications that have already been described in another closely related method. Currently, our approach does not deal with such scenarios as we do not consider contextual information. In future work, we plan to explore techniques to infer specifications in such scenarios.

Validation of Method Descriptions. API documents can sometimes be misleading [27], [33], thus causing developers to write faulty client code. In future work, we plan to extend our approach to find documentation-implementation inconsistencies.

Elimination of Predefined Lists. The current implementation of our approach uses predefined lists for domain dictionaries. There are approaches [41] that facilitate building domain dictionaries from source code. We plan to extend our implementation to use these approaches. Furthermore, we rely on pre-defined templates for code contract generation. While such a strategy serves our purpose of prototyping, advanced techniques such as keyword programming [18] have shown promising results in building programming statements using keywords. We plan to explore such techniques and evaluate the overall effectiveness of our approach after augmenting it with such techniques.

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], [21], [35]. However, a combination of developer written and automatically inferred contracts seems to be the most effective approach [8], [24]. 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. [37] 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 [12], [14], [15] or statically [3], [8] 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 [11], [28], [29], usability of API documents [6], and other areas [18], [41]. We next describe most relevant approaches.

Xiao et al. [39] 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.

Zhong et al. [40] 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. [33] 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.

VIII. Conclusion

Specifications described in natural language in API documents are not amenable to formal verification by existing verification tools. In this paper, we have presented a novel approach for inferring formal specifications from API documents targeted towards code contract generation. Our evaluation results show that our approach has an average of 92% precision and 93% recall in identifying sentences describing code contracts from over 2500 sentences. Furthermore, our results also show that our approach has an average of 83.4% accuracy in inferring specifications from sentences describing code contracts out of over 1600 sentences.

Acknowledgments

This work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNS-0958235, and ARO grant W911NF-08-1-0443. Hao Zhong is sponsored by the National Science Foundation of China under Grant No. 61100071.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proc. CASSIS, LNCS vol. 3362*, pages 49–69, Springer, 2004.

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

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

[4] P. Chalin. Are practitioners writing contracts? *The RODIN Book LNCS*, 4157(7):100, 2006.

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

[6] U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.

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

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

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

[10] J. Gaffney and R. Cruickshank. A general economics model of software reuse. In *Proc. 14th ICSE*, pages 327 – 337, 1992.

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

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

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

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

[15] 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.

[16] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. L. Pollock, and K. Vijay-Shanker. AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proc. MSR*, pages 79–88, 2008.

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

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

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

[20] B. Meyer. Applying 'design by contract'. *IEEE Transactions on Computer*, 25(10):40 –51, oct 1992.

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

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

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

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

[25] S. P. Reiss. Semantics-based code search. In *Proc. 31st ICSE*, pages 243–253, 2009.

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

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

[28] 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.

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

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

[31] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Proc. 37th POPL*, pages 313–326, 2010.

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

[33] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Proc. 5th ICST*, 2012.

[34] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. 22nd ASE*, pages 204–213, 2007.

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

[36] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proc. HLT-NAACL*, pages 252–259, 2003.

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

[38] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, C. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proc. 19th ISSTA*, pages 61–72, 2010.

[39] X. Xiao, A. Paradkar, and T. Xie. Automated extraction and validation of security policies from natural-language documents. Technical Report TR-2011-7, North Carolina State University Department of Computer Science, Raleigh, NC, March 2011.

[40] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. 24th ASE*, pages 307–318, November 2009.

[41] 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.