Inferring Semantic Information from Natural-Language Software Artifacts

Rahul Pandita
Department of Computer Science,
North Carolina State University, Raleigh, NC
Email:rpandit@ncsu.edu

A proposal for dissertation research submitted to the
Graduate Faculty of North Carolina State University
for the Oral-Preliminary Examination

November 26, 2013

Abstract

Code-level specifications play an important role in software engineering. In addition to guiding the development process by outlining what/how to reuse, specifications also help in verification process by allowing quality assurance practitioners to test the expected outcome. One of the valuable source of such specifications are the Natural language API documents. However, sometimes humans often overlook these documents and build software systems that are inconsistent with specifications described in those documents. While there are tools and frameworks available to assist humans to build/reuse quality software, these tools are not designed to work on specifications in natural language. To address this issue, this report presents a Natural Language Processing (NLP) framework to automate the task of inferring semantic information from natural language software artifacts to bridge the disconnect between the inputs required by software engineering tools/frameworks and the specifications described in natural language. This report is a part of a larger dissertation goal *to improve developer / tester / end-user productivity by automatically inferring semantic information from the textual descriptions in software artifacts*. Specifically, in this report I present two recent research efforts that I have conducted in developing/applying NLP techniques for inferring semantics from natural language software artifacts. Furthermore, I also outline my dissertation plan along with intermediate deliverables to achieve my goal.

1 Introduction

Specifications play an important role in software engineering. In general, specifications help end users to access the functionality of the software, before they actually use it. One such form of specifications are the code-level specifications. Not only, these specifications guide the development process by outlining what/how to reuse, they also help in verification process by allowing quality assurance practitioners to test the expected outcome.

While specifications in general are distributed across various software artifacts such as requirements documents, design documents and application descriptions. Application Programming Interface (API) documents, that are targeted towards developers, are invaluable source of information regrading code-level specifications. Typically, library developers commonly describe legal usage of the library in natural language text in API documents. Such documents are usually provided to client-code developers through online access, or are shipped with the API code. For example, J2EE's API documentation[1] is one of the popular API documents.

API documentation provides developers with useful information about class/interface hierarchies within the software. Additionally, API documents also provides information about how to use a particular method within a class by means of method descriptions. Method descriptions typically describe specifications in terms of the expectations of the method arguments (pre-conditions), expected return values (post-conditions) and functionality of method in general. These specifications are in valuable to automated tools directed towards improving developer/tester productivity. For instance, a typical automated testing tool will check to see if any of the pre-condition or the post condition is violated. However, such tools are not designed to work with the natural language descriptions, and often require more

[1] `http://download.oracle.com/javaee/1.6/api/`

formal specifications. Thus, there is a disconnect between the inputs required by these tools and the natural language specifications described in software artifacts.

The goal of my dissertation is to improve developer / tester / end-user productivity by automatically inferring the semantic information, in the form of formal specifications, from the textual descriptions in software artifacts. These formal specifications are meant as a means to bridge the gap between the existing tools that help the target audience and inputs required by tools.

In this report, I present two recent research efforts that I have conducted in developing/applying NLP techniques for discovering specifications out of NL software artifacts. These evaluation of these efforts show promising results to improve developer/tester productivity. For example, formally representing how to use an API method would assist a developer in writing better code. Similarly, a formal representation of appropriate usage of an API method will also benefit a quality assurance practitioner by explicitly delineating the illegal usages of an API. From an end users perceptive automated analysis of application description will assist them by answering a variety of questions regrading privacy and security such as: how a users private data will be used. I next present, two of my previous efforts. Particularly, I describe how the NLP is used to infer specifications from natural language software artifacts targeted towards assisting quality assurance practitioners and end users.

Assisting quality assurance practitioners: [2] Code contracts [43, 8] 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 (*pre-conditions*) and what to expect after method execution (*post-conditions*). Furthermore, code contracts can be subjected to formal verification by existing state-of-the-art verification tools such as Spec# [3], JML[3], and Code Contracts for .NET[4]. Additionally, code contracts can be used for formal proofs and automated code correction [69].

Despite being highly desirable, code contracts do not exist in a formalized form in most existing software systems in practice [50]. 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[5] 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 [45]. 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. 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.

***WHYPER* Bridging the gap between user expectations and software behavior:** [6] Application markets such as Apple's App Store and Google's Play Store have become the *de facto* mechanism of delivering software to consumer

[2]Note: The work presented here and in Sections 4- 5 is part of an academic research project done under supervision of my co-advisor Dr. Tao Xie (Associate Professor at Illinois) and collaborators Xusheng Xiao (NC State PhD Candidate), Hao Zhong, Stephan Oney and Amit Paradkar (External Collaborators), hence I use words `''our'', ''we''` instead of `''my'', ''I''` in these sections of this report. However, entire work presented in these sections (inclusive of writing this report) was carried out by me individually. These collaborators provided me with the timely guidance in terms of technical review of both ideas and writing

[3]`http://www.eecs.ucf.edu/~leavens/JML/`

[4]`http://research.microsoft.com/en-us/projects/contracts/`

[5]`http://download.oracle.com/javaee/1.6/api/`

[6]Note: The work presented here and in Sections 6- 7 is part of an academic research project done under supervision of my co-advisor Dr. Tao Xie (Associate Professor at Illinois), Dr. William Enck (Assistant Professor at NCSU), Xusheng Xiao (NC State PhD Candidate), and Wei Yang (UIUC graduate student), hence I use words `''our'', ''we''` instead of `''my'', ''I''` in these sections of this report. However, entire work presented in these sections (inclusive of writing this report) was carried out by me individually. These collaborators provided me with the timely guidance in terms of technical review of both ideas and writing, with the exception of student collaborators that helped me with evaluation.

smartphones and mobile devices. However, keeping malware out of these markets is an ongoing challenge.

It is non-trivial to classify an application as malicious, privacy infringing, or benign. Previous work has looked at permissions [18, 79, 49, 7], code [14, 16, 78, 27, 25], and runtime behavior [15, 36, 71]. However, underlying all of this work is a caveat: *what does the user expect?* Clearly, an application such as a *GPS Tracker* is expected to record and send the phone's geographic location to the network; an application such as a *Phone-Call Recorder* is expected to record audio during a phone call; and an application such as *One-Click Root* is expected to exploit a privilege-escalation vulnerability. Other cases are more subtle. The Apple and Google approaches fundamentally differ in who determines whether an application's permission, code, or runtime behavior is appropriate. For Apple, it is an employee; for Google, it is the end user.

We are motivated by the vision of bridging the semantic gap between what the user expects an application to do and what it actually does. This work is a first step in this direction. Specifically, we focus on permissions and ask the question, *does the application description provide any indication for the application's use of a permission?* Clearly, this hypothesis will work better for some permissions than others. For example, permissions that protect a user-understandable resource such as the address book, calendar, or microphone should be discussed in the application description. However, other low-level system permissions such as accessing network state and controlling vibration are not likely to be mentioned. We note that while this work primarily focuses on permissions in the Android platform and relieving the strain on end users, it is equally applicable to other platforms (e.g., Apple) by aiding the employee performing manual inspection.

With this vision, in this paper, we present *WHYPER*, a framework that uses Natural Language Processing (NLP) techniques to determine *why* an application uses a *per*mission. *WHYPER* takes as input an application's description from the market and a semantic model of a permission, and determines which sentence (if any) in the description indicates the use of the permission. Furthermore, we show that for some permissions, the permission semantic model can be automatically generated from platform API documents. We evaluate *WHYPER* against three popularly-used permissions (address book, calendar, and record audio) and a dataset of 581 popular applications. These three frequently-used permissions protect security and privacy sensitive resources. Our results demonstrate that *WHYPER* effectively identifies the sentences that describe needs of permissions with an average precision of 82.8% and an average recall of 81.5%. We further investigate the sources of inaccuracies and discuss techniques of improvement.

The rest of the report is organized as follows. Section 2 presents the background on concepts used in this work. Section 3 discusses related work. Section 4 presents our approach to infer method specifications from API documents, followed by evaluation in Section 5. Section 6 presents our *WHYPER* approach, followed by evaluation in Section 7. Section 8 presents a dissertation goals, future work, and planned time-line of achieving goals. Finally, Section 9 concludes.

2 Background

This section briefly introduces the technologies/methodologies used in this report. In particular, the section provides background information on code contracts, NLP techniques used in this report, and the first-order-logic representation for the specifications.

2.1 Code Contracts

Code contracts, based on the Design by Contracts (DbC) [8] 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 [69].

2.2 NLP Preliminaries

Natural language is well suited for human communication, but converting natural language into unambiguous specifications that can be processed and understood by computers is difficult. However, research advances [10, 11, 39, 40]

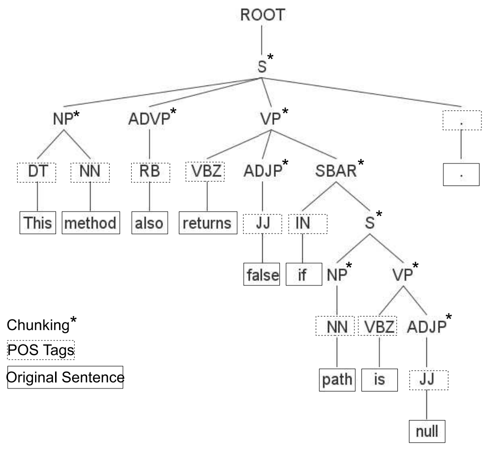

Figure 1: An example of POS tagging for comment “This method also returns false if path is null”

have increased the accuracy of existing NLP techniques to annotate the grammatical structure of a sentence. We next introduce the core NLP techniques used in this work.

- **Parts Of Speech (POS) tagging** [39, 40]. 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% [57] 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% [57] accuracy in classifying phrases and clauses over well written news articles.

- **Typed Dependencies** [10, 11]. 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** [19]. 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** [51, 41]. 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.

2.3 Formal representation used in this work

We use First-Order Logic (FOL) expressions for representing the specifications inferred from the natural language text in API documents. We chose FOL as formal representation because the syntax is simple and yet powerful enough to represent the inferred specifications.

Terms and formulas form the basic building blocks of FOL expressions. Terms can be further resolved into two sub categories: variables and functions. Functions constitutes of arguments that are terms. Thus, arguments can be either variables or functions. Below are the five rules for constructing valid FOL expressions:

- *Predicate Symbols*: If P is a Predicate symbol involving n terms then $P(t_1, t_2, ...t_n)$ is a valid FOL expression.

- *Equality*: If t_1 and t_2 are two valid terms then $t_1 = t_2$ is valid FOL expression.
- *Negation*: If E_1 is a valid FOL expression then its negation, denoted by $\neg E$ is a valid FOL expression.
- *Binary Connectives*: If E_1 and E_2 are two valid FOL expressions then an expression involving binary connectives (inclusive of $\wedge, \vee, and \rightarrow$) is a valid FOL expression. The symbol $\wedge$ is 'conjunction', $\vee$ is 'disjunction' and $\rightarrow$ is a conditional (if/then) statement.
- *Quantifiers*: If E is a valid FOL expression and v is a valid variable then $\exists v E$ and $\forall v E$ are valid expressions.

3 Related Work

Our proposed techniques touches quiet a few research areas such as NLP on software engineering artifacts, program synthesis and software verification. We next discuss relevant work pertinent to our proposed framework in these areas.

Code Contracts

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 [9, 44, 66]. However, a combination of developer written and automatically inferred contracts seems to be the most effective approach [50, 20]. 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. [68] 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[33, 24, 34] or statically [20, 6] from source code and binaries. In contrast, the approach presented in this work infers specifications from the natural language text in API documents, thus complementing these existing approaches when the source code or binaries of the API library is not available.

NLP in Software Engineering

NLP techniques are increasingly applied in the software engineering domain. NLP techniques have been shown to be useful in requirements engineering [54, 55, 23], usability of API documents [12], and other areas [77, 42]. We next describe most relevant approaches.

- *Access control policies*: Xiao et al. [70] and Slankas et al. [56] 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.
- *Resource Specifications*: Zhong et al. [76] employ NLP and Machine Learning (ML) techniques to infer resource specifications from API documents. Their approach uses machine learning to automatically classify such rules. In contrast, we attempt to parse sentences based on semantic templates and demonstrate that such an approach preforms reasonably well. Furthermore, the performance of the approaches is dependent on the quality of the training sets used for ML. In contrast, approach presented in this work is independent of such training set and thus can be easily extended to target respective problems addressed by these approaches.
- *Code Comments*: Tan et al. [62] applied an NLP and ML based approach on code comments to detect mismatches between these comments and implementations. They rely on predefined rule templates targeted towards method invocation and lock related comments, thus limiting their scope both in terms of application area as well as language used in the comments. In contrast, approach presented in this report relies on generic natural language based templates thus relaxing the restriction on the style of the language used to describe specifications.
- *Javadoc Comments*: Hwei-Tan et al. [63] extended the work of Tan et al. [62] to apply NLP and ML based approach to test Javadoc comments against implementations. However, the approach specifically focuses on null values and related exceptions, thus limiting the application scope. In contrast, approach presented in this report infers generic specifications from API documents. Furthermore, approach presented in this report already produces FOL representation of the specifications that can be used to test implementation.

- *Requirements Document*:Sinha et al. [54, 55] used NLP on natural language text appearing in the use cases for automated and "edit-time" inspection of use cases based on the construction and analyses of models. In particular the model construction allows them to perform variety of checks such as: stylistic checks for the language, complexity checks for number of actions being performed in a use case, flow checks for use of an item before it is created... In contrast, the approaches presented in this report work on more concrete code level specifications. We believe the techniques presented in this work can further improve their model constructions and their work can further work in conjunction to the techniques presented in this work to infer better specifications.

Program Comprehension

With respect to program comprehension there are existing techniques that assist in building domain specific ontologies [77]. Furthermore, there are existing approaches [58, 52] that automatically infer natural language documentation from source code. These approaches would immensely help in comprehension of the functionality of an application. However inherent dependency on source code to generate such documents poses a problem in cases, where source code is not available. We next describe some of these approaches

- Sridhara et al. [58] use data-flow analysis on the method variables to determine the set of statements that represent the computational intent of the method. The data flow analysis is then used to determine the relation of these statements to the method parameters. These realtions are then thn used to generate the natural language parameter documentation using predefined templates.

- Zhang et al. [72] present an approach to generate the explanatory document in the form of code comments explaining the failure of the test case. They first instrument the failing test case. They then use statistical algorithm on the different execution traces of the failing test case to determine a relatively small subset of the suspicious statements along with the objects that need correction for successful execution of test case. Finally, the identified objects that need correction for successful execution of the test case are used to generate the explanatory comments using predefined templates on the generalized properties of the object. For instance, `x == null` becomes *x is set to:null*.

- Buse et al. [6] use source mining source code repositories to generate usage patterns of an API. They argue that availability of actual usable code examples serves as a better documentation assisting developers in understating the usage of API. They propose the clustering of the uses of an API in the open source repositories based on path information. They then propose type abstraction on the clusters to come up with a usage document.

Program Synthesis

Automated program synthesis has been gaining traction with a lot of recent work [29, 30, 32, 31, 37, 38, 59, 61, 64] in this direction. Srivastava et. al. [59] addresses the problem by leveraging specifications in the form of pre/post conditions and invariant to achieve synthesis. There is also some work [32, 37, 61, 38] in literature that addresses the problem of program synthesis by leveraging the state of the art constraint solving. However, there is disconnect between the synthesis achieved by these approaches and inputs required by them to achieve synthesis. Most of these approaches accept either precise formal specifications in the form of pre-post conditions or extensive input-output data. The synthesis achieved providing input-output relationships is sensitive to the the input data provided. In contrast, precise formal specifications while highly desirable, are rarely found in practice. Even the formal requirements and design documents of almost all of the projects are invariably mixed with natural language sentences.

Among these approaches Thummalapenta et. al. [64] work is the closest to the techniques proposed in this report. They propose techniques to automate the manual test case, which is sequence of steps written in natural language. in particular, they propose to use a novel combination of NLP along with backtracking exploration, runtime interpretation to guide creation of the automated test scripts.

API Mapping

There has been some work [26, 74, 73] on determining how the methods in a source API maps to methods in target API. Zhong et al. [74], propose static analysis based approach to determine method mapping across API. Their approach requires an application that has been manually translated from source API to an target API. These applications are then subjected to static analysis and name similarity to *align* the classes and methods in source API to target API. Such *aligned* classes and methods are assumed to have similar functionality.

Gokhale et al. [26] relaxes the constraints on the having applications manually ported before static analysis. In contrast, they use *similar* applications in source and target platform. Two applications are considered *similar* if they

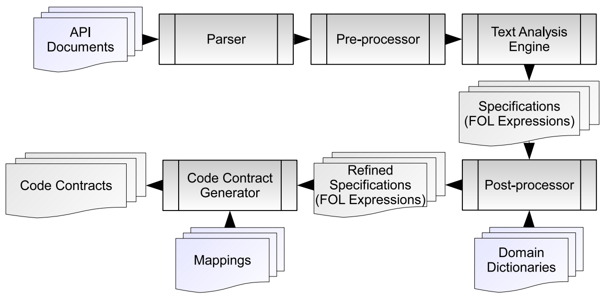

Figure 2: Overview of our approach

have provide same core functionality. These similar applications are first instrumented and then executed with identical inputs. During the execution the sequence of calls to underlying platform API are recorded. Finally, static analysis techniques are applied on these sequence of platform API calls to determine the mapping of methods across source and target platform API.

In contrast, Zheng et al. [73] use an experimental web search based approach to determining such mapping. First they formulate a query based on the name of the method in source API along with the name of target API. They then use the query to search on a web search engine. Resultant top-k web pages are then subjected to text analysis, specifically looking for method names in target API. These methods are finally presented to the user as a possible mapping along with contents of the web page for user to make informed decision.

4 Approach for inferring method specifications from API documents

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.

4.1 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.

4.2 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

Table 1: 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.

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 [35] 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.

4.3 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 [54, 55] 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

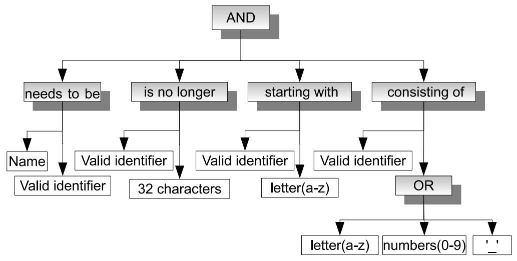

Figure 3: Specifications in format of FOL expressions extracted by our NLP Parser for `DefineObjectProperty` method in Facebook API

```
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

parsing [4]. 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 [4, 60, 54, 28] 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 1 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 1, 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 1 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 [57], 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.

4.4 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.

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

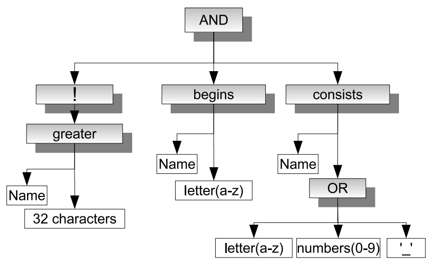

Figure 5: FOL expression after synonym analysis and compaction for the `DefineObjectProperty` method in Facebook API

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 [57]. 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 [17] 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 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

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 $(e'.root ==$ “ $\rightarrow$ ”$)\&\&(e'.right\ is\ variable)$ then
 4:         $Term\ t = e'.right$
 5:         if $findType(t) == d.returnType$ then
 6:             $Predicate\ p = new\ Predicate(“returns”)$
 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 $(e'.root ==$ “ $\rightarrow$ ”$)\&\&(e'.right\ is\ empty)$ then
14:         $Term\ t = d.exception\_name$
15:         $Predicate\ p = new\ Predicate\ (“throw”)$
16:         $p.term = t$
17:         $e'.right = p$
18:     end if
19:     if $(e'.root! ==$ “ $\rightarrow$ ”$)$ then
20:         $Term\ t = d.exception\_name$
21:         $Predicate\ p = new\ Predicate\ (“throw”)$
22:         $p.term = t$
23:         $Expr\ e'' = new\ Expr($“ $\rightarrow$ ”$)$
24:         $e''.left = e'$
25:         $e''.right = p$
26:         $e' = e''$
27:     end if
28: end if
29: return $e'$
```

```
requires(!prop_name.length()>32)
requires(prop_name.substring(0,1).
   matches([a-z]+))
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

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.

4.5 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]+"))`.

Table 2: 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

5 Evaluation of method specification inference from API

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?

5.1 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 2 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.

5.2 Evaluation Results

We next describe our evaluation results to demonstrate the effectiveness of proposed approach in identifying contract sentences.

5.2.1 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.

[7]`https://www.facebook.com/press/info.php?statistics`
[8]`http://facebooktoolkit.codeplex.com.`

- **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 [46], 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 2 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. 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.

5.2.2 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 2 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%.

5.2.3 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. [53] 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 2 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"* [53]. 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 [9, 44].

5.2.4 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 2 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 2 from over 1600 contract sentences. Furthermore, our evaluation results show that our approach can infer detailed specifications than human written contracts.

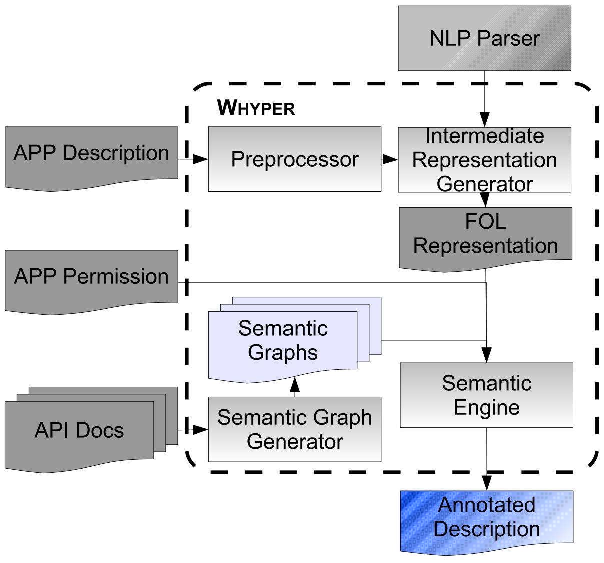

Figure 7: Overview of WHYPER framework

5.3 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# [53], 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.

6 WHYPER Design

We next present our framework for annotating the sentences that describe the needs for permissions in application descriptions. Figure 7 gives an overview of our framework. Our framework consists of five components: a preprocessor, an NLP Parser, an intermediate-representation generator, a semantic engine (SE), and an analyzer.

The pre-processor accepts application descriptions and preprocesses the sentences in the descriptions, such as annotating sentence boundaries and reducing lexical tokens. The intermediate-representation generator accepts the pre-processed sentences and parses them using an NLP parser. The parsed sentences are then transformed into the first-order-logic (FOL) representation. SE accepts the FOL representation of a sentence and annotates the sentence based on the semantic graphs of permissions. Our semantic graphs are derived by analyzing Android API documents. We next describe each component in detail.

6.1 Preprocessor

The preprocessor accepts natural-language application descriptions and preprocesses the sentences, to be further analyzed by the intermediate-representation generator. The preprocessor annotates sentence boundaries, and reduces the number of lexical tokens using semantic information. The reduction of lexical tokens greatly increases the accuracy of the analysis in the subsequent components of our framework. In particular, the preprocessor performs following preprocessing tasks:

1. Period Handling. In simplistic English, the character period (‘.’) marks the end of a sentence. However, there are other legal usages of the period such as: (1) decimal (periods between numbers), (2) ellipsis (three continuous periods ‘...’), (3) shorthand notations (“Mr.”, “Dr.”, “e.g.”). While these are legal usages, they 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 [17] 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. Sentence Boundaries. Furthermore, there are instances where an enumeration list is used to describe functionality, such as *“The app provides the following functionality: a) abc..., b) xyz... ”*. While easy for a human to understand the meaning, it is difficult from a machine to find appropriate boundaries.

We leverage the structural (positional) information: (1) placements of tabs, (2) bullet points (numbers, characters, roman numerals, and symbols), and (3) delimiters such as “:” to detect appropriate boundaries. We further improve the boundary detection using the following patterns we observe in application descriptions:

- We remove the leading and trailing ‘*’ and ‘-’ characters in a sentence.
- We consider the following characters as sentence separators: ‘–’, ‘- ’, ‘ø’, ‘§’, ‘†’, ‘⋄’, ‘♢’, ‘♣’, ‘♡’, ‘♠’ ... A comprehensive list can be found on the project website [1].
- For an enumeration sentence that contains at least one enumeration phrase (longer than 5 words), we break down the sentence to short sentences for each enumerated item.

3. 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 *“Pandora internet radio”*, *“Google maps”*, which are the names of applications. Further resolution of these phrases using grammatical syntax is unnecessary and would not bring forth any semantic value. Thus, we identify such phrases and annotate them as single lexical units. We achieve so by maintaining a static lookup table.

4. 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 *“Instant Message (IM)”* 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.

6.2 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 [57]. However, this component can be implemented using any other existing NLP libraries or frameworks:

1. **Named Entity Recognition:** NLP parser identifies the named entities in the document and annotates them. Additionally, these entities are further added to the lookup table, so that the preprocessor use the entities for processing subsequent sentences.
2. **Stanford-Typed Dependencies:** [10, 11] NLP parser further annotates the sentences with Stanford-typed dependencies. Stanford-typed dependencies is a simple description of the grammatical relationships in a sentence, and targeted towards extraction of textual relationships. In particular, we use standford-typed dependencies as an input to our intermediate-representation generator.

Next we use an example to illustrate the annotations added by the NLP Parser. Consider the example sentence ***“Also you can share the yoga exercise to your friends via Email and SMS.”***, that indirectly refers to the READ_CONTACTS permission. Figure 8 shows the sentence annotated with Stanford-typed dependencies. The words in red are the names of dependencies connecting the actual words of the sentence (in black). Each word is followed by the Part-Of-Speech (POS) tag of the word (in green). For more details on Stanford-typed dependencies and POS tags, please refer to [10, 11].

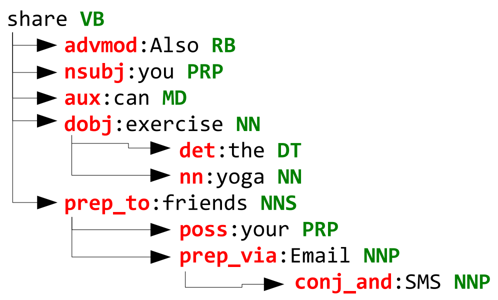

Figure 8: Sentence annotated with Stanford dependencies

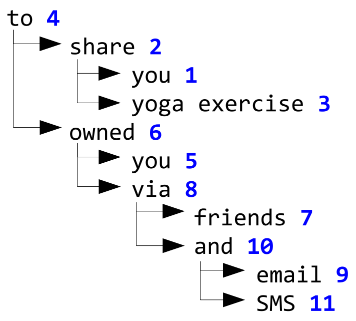

Figure 9: First-order logic representation of annotated sentence in Figure 8

6.3 Intermediate-Representation Generator

The intermediate-representation generator accepts the annotated documents and builds a relational representation of the document. We define our representation as a tree structure that is essentially a First-Order-Logic (FOL) expression. Recent research has shown the adequacy of using FOL for NLP related analysis tasks [54, 55, 48]. 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.

We implemented our intermediate-representation generator based on the principle shallow parsing [4] techniques. A typical shallow parser attempts to parse a sentence based on the function of POS tags. However, we implemented our parser as a function of Stanford-typed dependencies [10, 11, 57, 40]. We chose Stanford-typed dependencies for parsing over POS tags because Stanford-typed dependencies annotate the grammatical relationships between words in a sentence, thus provide more semantic information than POS tags that merely highlight the syntax of a sentence.

In particular, our intermediate-representation generator is implemented as a series of cascading finite state machines (FSM). Earlier research [4, 60, 54, 28, 48] has shown the effectiveness and efficiency of using FSM in linguistics analysis such as morphological lookup, POS tagging, phrase parsing, and lexical lookup. We wrote semantic templates for each of the typed dependencies provided by the Stanford Parser.

Table 3: Semantic Templates for Stanford Typed Dependencies

S. No.	Dependency	Example	Description
1	**conj**	*"Send via SMS and email."* **conj_and**(email, SMS)	Governor (**SMS**) and dependent (**email**) are connected by a relationship of conjunction type(**and**)
2	**iobj**	*"This App provides you with beautiful wallpapers."* **iobj**(provides, you)	Governor (**you**) is treated as dependent entity of relationship resolved by parsing the dependent's (**provides**) typed dependencies
3	**nsubj**	*"This is a scrollable widget."* **nsubj**(widget, This)	Governor (**This**) is treated as governing entity of relationship resolved by parsing the dependent's (**widget**) typed dependencies

Table 3 shows a few of these semantic templates. Column “Dependency” lists the name of the Stanford-typed dependency, Column “Example” lists an example sentence containing the dependency, and Column “Description” describes the formulation of tuple from the dependency. All of these semantic templates are publicly available on our project website [1]. Figure 9 shows the FOL representation of the sentence in Figure 8. For ease of understanding, read the words in the order of the numbers following them. For instance, *“you”* $\leftarrow$ *“share”* $\rightarrow$ *“yoga exercise”* forms one tuple. Notice the additional predicate “owned” annotated 6 in the Figure 9, does not appear in actual sentence. The additional predicate “owned” is inserted when our intermediate-representation generator encounters the possession modifier relation (annotated “poss” in Figure 8).

The FOL representation helps us effectively deal with the problem of *confounding effects* of keywords. In particular, the FOL assists in distinguishing between a resource that would be a leaf node and an action that would be a predicate node in the intermediate representation of a sentence. The generated FOL representation of the sentence is then provided as an input to the semantic engine for further processing.

6.4 Semantic Engine (SE)

The Semantic Engine (SE) accepts the FOL representation of a sentence and based on the semantic graphs of Android permissions annotates a sentence if it matches the criteria. A semantic graph is basically a semantic representation of the resources which are governed by a permission. For instance, the READ_CONTACTS permission governs the resource “CONTACTS” in Android system.

Figure 10 shows the semantic graph for the permission READ_CONTACTS. A semantic graph primarily constitutes of subordinate resources of a permission (represented in rectangular boxes) and a set of available actions on the resource itself (represented in curved boxes). Section 6.5 elaborates on how we build such graphs systematically.

Our SE accepts the semantic graph pertaining to a permission and annotates a sentence based on the algorithm shown in Algorithm 2. The Algorithm accepts the FOL representation of a sentence rep, the semantic graph associated with the resource of a permission g and a boolean value $recursion$ that governs the recursion. The algorithm outputs a boolean value $isPStmt$, which is `true` if the statement describes the permission associated with a semantic graph (g), otherwise `false`.

Our algorithm systematically explores the FOL representation of the sentence to determine if a sentence describes the need for a permission. First, our algorithm attempts to locate the occurrence of associated resource name within the leaf node of the FOL representation of the sentence (Line 3). The method `findLeafContaining(name)` explores the FOL representation to find a leaf node that contains term `name`. Furthermore, we use WordNet and Lemmatisation [13] to deal with synonyms of a word in question to find appropriate matches. Once a leaf node is found, we systematically traverse the tree from the leaf node to the root, matching all parent predicates as well as immediate child predicates [Lines 5-16].

Our algorithm matches each of the traversed predicate with the actions associated with the resource defined in semantic graph. Similar to matching entities, we also employ WordNet and Lemmatisation [13] to deal with synonyms to find appropriate matches. If a match is found, then the value `isPStmt` is set to `true`, indicating that the statement describes a permission.

In case no match is found, our algorithms recursively search all the associated subordinate resources in the semantic graph of current resource. A subordinate resource may further have its own subordinate resources. Currently, our algorithm considers only immediate subordinate resources of a resource to limit the false positives.

In the context of the FOL representation shown in Figure 9, we invoke Algorithm 2 with the semantic graph shown in Figure 10. Our algorithm attempts to find a leaf node containing term “CONTACT” or some of its synonym. Since the FOL representation does not contain such a leaf node, algorithm calls itself with semantic graphs of subordinate resources (Line 17-25), namely ‘NUMBER’, ‘EMAIL’, ‘LOCATION’, ‘BIRTHDAY’, ‘ANNIVERSARY’.

The subsequent invocation will find the leaf-node “email” (annotated 9 in Figure 9). Our algorithm then explores the preceding predicates and finds predicate “share” (annotated 2 in Figure 9). The Algorithm matches the word “share” with action “send” (using Lemmatisation and WordNet similarity), one of the actions available in the semantic graph of resource ‘EMAIL’ and returns `true`. Thus, the sentence is appropriately identified as describing the need for permission READ_CONTACT.

Algorithm 2 Sentence_Annotator

```
Input: K_Graph $g$, FOL_rep $rep$, Boolean $recursion$
Output: Boolean $isPStmt$
 1: $Boolean\ isPStmt\ =\ false$
 2: $String\ r\_name\ =\ g.resource\_Name$
 3: $FOL\_rep\ r'\ =\ rep.findLeafContaining(r\_name)$
 4: $List\ actionList\ =\ g.actionList$
 5: while $(r'.hasParent)$ do
 6:     if $actionList.contains(r'.parent.predicate)$ then
 7:         $isPStmt\ =\ true$
 8:         $break$
 9:     else
10:         if $actionList.contains(r'.leftSibling.predicate)$ then
11:             $isPStmt\ =\ true$
12:             $break$
13:         end if
14:     end if
15:     $r'\ =\ r'.parent$
16: end while
17: if $((NOT(isPStmt))\ AND\ recursion)$ then
18:     $List\ resourceList\ =\ g.resourceList$
19:     for all $(Resource\ res\ in\ resourceList)$ do
20:         $isPStmt\ =\ Sentence\_Annotator(getKGraph(res),\ rep,\ false)$
21:         if $isPStmt$ then
22:             $break$
23:         end if
24:     end for
25: end if
26: return $isPStmt$
```

6.5 Semantic-Graph Generator

A key aspect of our proposed framework is the employment of a semantic graph of a permission to perform deep semantic inference of sentences. In particular, we propose to initially infer such graphs from API documents. For third-party applications in mobile devices, the relatively limited resources (memory and computation power compared to desktops and servers) encourage development of thin clients. *The key insight to leverage API documents is that mobile applications are predominantly thin clients, and actions and resources provided by API documents can cover most of the functionality performed by these thin clients.*

Manually creating a semantic graph is prohibitively time consuming and may be error prone. We thus came up with a systematic methodology to infer such semantic graphs from API documents that can potentially be automated. First, we leverage Au et al.'s work [2] to find the API document of the class/interface pertaining to a particular permission. Second, we identify the corresponding resource associated with the permission from the API class name. For instance, we identify 'CONTACTS' and 'ADDRESS BOOK' from the `ContactsContract.Contacts`[9] class that is associated with READ_CONTACT permission. Third, we systematically inspect the member variables and member methods to identify actions and subordinate resources.

From the name of member variables, we extract noun phrases and then investigate their types for deciding whether these noun phrases describe resources. For instance, one of member variables of `ContactsContract.Contracts` class leads us to its member variable "email" (whose type is `ContactsContract.CommonDataKinds.Email`). From this variable, we extract noun phrase "EMAIL" and classify the phrase as a resource.

From the name of an Android API public method (describing a possible action on a resource), we extract both noun phrases and their related verb phrases. The noun phrases are used as resources and the verb phrases are used as the associated actions. For instance, `ContactsContract.Contacts` defines operations Insert, Update, Delete, and so on. We consider those operations as individual actions associated with 'CONTACTS' resource.

The process is iteratively applied to the individual subordinate resources that are discovered for an API. For instance, "EMAIL" is identified as a subordinate resource in "CONTACT" resource. Figure 10 shows a sub-graph of graph for READ_CONTACT permission.

[9]`http://developer.android.com/reference/android/provider/ContactsContract.Contacts.html`

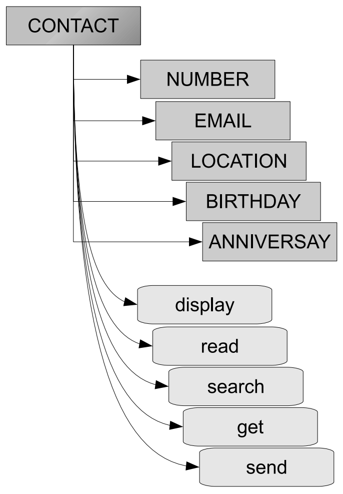

Figure 10: Semantic Graph for the READ_CONTACT permission

7 WHYPER Evaluation

We now present the evaluation of WHYPER. Given an application, the WHYPER framework bridges the semantic gap between user expectations and the permissions it requests. It does this by identifying in the application description the sentences that describe the need for a given permission. We refer to these sentences as *permission sentences*. To evaluate the effectiveness of WHYPER, we compare the permission sentences identified by WHYPER to a manual annotation of all sentences in the application descriptions. This comparison provides a quantitative assessment of the effectiveness of WHYPER. Specifically, we seek to answer the following research questions:

- **RQ1**: What are the precision, recall and F-Score of WHYPER in identifying permission sentences (i.e., sentences that describe need for a permission)?
- **RQ2**: How effective WHYPER is in identifying permission sentences, compared to keyword-based searching ?

7.1 Subjects

We evaluated WHYPER using a snapshot of popular application descriptions. This snapshot was downloaded in January 2012 and contained the top 500 free applications in each category of the Google Play Store (16,001 total unique applications). We then identified the applications that contained specific permissions of interest.

For our evaluation, we consider the READ_CONTACTS, READ_CALENDAR, and RECORD_AUDIO permissions. We chose these permissions because they protect tangible resources that users understand and have significant enough security and privacy implications that developers should provide justification in the application's description. We found that 2327 applications had at least one of these three permissions. Since our evaluation requires manual effort to classify each sentence in the application description, we further reduced this set of applications by randomly choosing 200 applications for each permission. This resulted in a total of 600 unique applications for these permissions. The set was further reduced by only considering applications that had an English description). Overall, we analysed 581 application descriptions, which contained 9,953 sentences (as parsed by WHYPER).

7.2 Evaluation Setup

We first manually annotated the sentences in the application descriptions. We had three authors independently annotate sentences in our corpus, ensuring that each sentence was annotated by at least two authors. The individual annotations were then discussed by all three authors to reach to a consensus. In our evaluation, we annotated a sentence as a permission sentence if at least two authors agreed that the sentence described the need for a permission. Otherwise we annotated the sentence as a permission-irrelevant sentence.

Table 4: Statistics of Subject permissions

Permission	$\#N$	$\#S$	S_P
READ_CONTACTS	190	3379	235
READ_CALENDAR	191	2752	283
RECORD_AUDIO	200	3822	245
TOTAL	581	9953	763

#N: Number of applications that requests the permission; #S: Total number of sentences in the application descriptions; S_P: Number of sentences manually identified as permission sentences.

We applied WHYPER on the application descriptions and manually measured the number of `true positives` (TP), `false positives` (FP), `true negatives` (TN) and `false negatives` (FN) produced by WHYPER as follows:

1. TP: A sentence that WHYPER correctly identifies as a permission sentence.
2. FP: A sentence that WHYPER incorrectly identifies as a permission sentence.
3. TN: A sentence that WHYPER correctly identifies as not a permission sentence.
4. FN: A sentence that WHYPER incorrectly identifies as not a permission sentence.

Table 4 shows the statistics of the subjects used in the evaluations of WHYPER. Column "Permission" lists the names of the permissions. Column "$\#N$" lists the number of applications that requests the permissions used in our evaluations. Column "#S" lists the total number of sentences in application descriptions. Finally, Column "S_P" lists the number of sentences that are manually identified as permission sentences by authors. We used this manual identification (Column "S_P") to quantify the effectiveness of WHYPER in identifying permission sentences by answering RQ1. The results of Column "S_P" is also used to compare WHYPER with keyword-based searching to answer RQ2 described next.

For RQ2, we applied keyword-based searching on the same subjects. We consider a word as a keyword in the context of a permission if it is a synonym of the word in the permission. To minimize manual efforts, we used words present in `Manifest.permission class` from Android API. Table 6 shows the keywords used in our evaluation. We then measured the number of `true positives` (TP'), `false positives` (FP'), `true negatives` (TN'), and `false negatives` (FN') produced by the keyword-based searching as follows:

1. TP':- A sentence that is a permission sentence and contains the keywords.
2. FP':- A sentence that is not a permission sentence but contains the keywords.
3. TN':- A sentence that is not a permission sentence and does not contain the keywords.
4. FN':- A sentence that is a permission sentence but does not contain the keywords.

In statistical classification [46], *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. *Accuracy* is defined as the ratio of sum of true positives and true negatives to the total number of items. Higher values of precision, recall, F-Score, and accuracy indicate higher quality of the permission sentences inferred using WHYPER. Based on the total number of TPs, FPs, TNs, and FNs, we computed the precision, recall, F-score, and accuracy of WHYPER in identifying permission sentences as follows:

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

Table 5: Evaluation results

Permission	S_I	TP	FP	FN	TN	P (%)	R (%)	F_S (%)	Acc (%)
READ_CONTACTS	204	186	18	49	2930	91.2	79.1	84.7	97.9
READ_CALENDAR	288	241	47	42	2422	83.7	85.1	84.4	96.8
RECORD_AUDIO	259	195	64	50	3470	75.9	79.7	77.4	97.0
TOTAL	751	622	129	141	9061	82.8*	81.5*	82.2*	97.3*

* Column average; S_I: Number of sentences identified by WHYPER as permission sentences; TP: Total number of True Positives; FP: Total number of False Positives; FN: Total number of False Negatives; TN: Total number of True Negatives; P: Precision; R: Recall; F_S: F-Score; and Acc: Accuracy

7.3 Results

We next describe our evaluation results to demonstrate the effectiveness of WHYPER in identifying permission sentences.

7.3.1 RQ1: Effectiveness in identifying permission sentences

In this section, we quantify the effectiveness of WHYPER in identifying permission sentences by answering RQ1. Table 5 shows the effectiveness of WHYPER in identifying permission sentences. Column "Permission" lists the names of the permissions. Column "S_I" lists the number of sentences identified by WHYPER as permission sentences. Columns "TP", "FP", "TN", and "FN" represent the number of `true positives`, `false positives`, `true negatives`, and `false negatives`, respectively. Columns "P(%)", "R(%)", "F_S(%)", and "Acc(%)" list percentage values of `precision`, `recall`, `F-score`, and `accurary` respectively. Our results show that, out of 9,953 sentences, WHYPER effectively identifies permission sentences with the average precision, recall, F-score, and accuracy of 82.8%, 81.5%, 82.2%, and 97.3%, respectively.

We also observed that out of 581 applications whose descriptions we used in our experiments, there were only 86 applications that contained at least one false negative statement that were annotated by WHYPER. Similarly, among 581 applications whose descriptions we used in our experiments, there were only 109 applications that contained at least one false positive statement that were annotated by WHYPER.

We next present an example to illustrate how WHYPER incorrectly identifies a sentence as a permission sentence (producing false positives). False positives are particularly undesirable in the context of our problem domain, because they can mislead the users of WHYPER into believing that a description actually describes the need for a permission. Furthermore, an overwhelming number of false positives may result in user fatigue, and thus devalue the usefulness of WHYPER.

Consider the sentence "*You can now turn recordings into ringtones.*". The sentence describes the application functionality that allows users to create ringtones from previously recorded sounds. However, the described functionality does not require the permission to record audio. WHYPER identifies this sentence as a permission sentence. WHYPER correctly identifies the word *recordings* as a resource associated with the record audio permission. Furthermore, our intermediate representation also correctly constructs that action *turn* is being performed on the resource *recordings*. However, our semantic engine incorrectly matches the action *turn* with the action *start*. The later being a valid semantic action that is permitted by the API on the resource *recording*. In particular, the general purpose WordNet-based Similarity Metric [13] shows 93.3% similarity. We observed that a majority of false positives resulted from incorrect matching of semantic actions against a resources. Such instances can be addressed by using domain-specific dictionaries for synonym analysis.

Another major source of FPs is the incorrect parsing of sentences by the underlying NLP infrastructure. For instance, consider the sentence "*MyLink Advanced provides full synchronization of all Microsoft Outlook emails (inbox, sent, outbox and drafts), contacts, calendar, tasks and notes with all Android phones via USB.*". The sentence describes the users calendar will be synchronized. However, the underlying Stanford parser [**?**] is not able to accurately annotate the dependencies. Our intermediate-representation generator uses shallow parsing that is a function of these dependencies. An inaccurate dependency annotation causes an incorrect construction of intermediate representation and eventually causes an incorrect classification. Such instances will be addressed with the advancement in underlying NLP infrastructure. Overall, a significant number of false positives will be reduced by as the current NLP research advances the underlying NLP infrastructure.

We next present an example to illustrate how WHYPER fails to identify a valid permission sentence (false nega-

Table 6: Keywords for Permissions

S. No	Permission	Keywords
1	READ_CONTACTS	contact, data, number, name, email
2	READ_CALENDAR	calendar, event, date, month, day, year
3	RECORD_AUDIO	record, audio, voice, capture, microphone

tives). Consider the sentence "*Blow into the mic to extinguish the flame like a real candle*". The sentence describes a semantic action of blowing into the microphone. The noise created by blowing will be captured by the microphone, thus implying the need for record audio permission. WHYPER correctly identifies the word *mic* as a resource associated with the record audio permission. Furthermore, our intermediate representation also correctly shows that the action *blow into* is performed on the resource *mic*. However, from API documents, there is no way for WHYPER framework to infer the knowledge that blowing into microphone semantically implies recording of the sound. Thus, WHYPER fails to identify the sentence as a permission sentence. We can reduce a significant number of false negatives by constructing better semantic graphs.

Similar to reasons for false positives, a major source of false negatives is the incorrect parsing of sentences by the underlying NLP infrastructure. For instance, consider the sentence "*Pregnancy calendar is an application that,not only allows, after entering date of last period menstrual,to calculate the presumed (or estimated) date of birth; but, offering prospects to show,week to week,all appointments which must to undergo every mother,ad a rule,for a correct and healthy pregnancy.*" [10] The sentence describes that the users calendar will be used to display weekly appointments. However, the length and complexity in terms of number of clauses causes the underlying Stanford parser [**?**] to inaccurately annotate the dependencies, which eventually results into incorrect classification.

We also observed that in a few cases, the process followed to identify sentence boundaries resulted in extremely long and possibly incorrect sentences. Consider a sentence that our preprocessor did not identify as a permission sentence for READ_CALENDER permission:

Daily Brief "How does my day look like today" "Any meetings today" "My reminders" "Add reminder" Essentials Email, Text, Voice dial, Maps, Directions, Web Search "Email John Subject Hello Message Looking forward to meeting with you tomorrow" "Text Lisa Message I will be home in an hour" "Map of Chicago downtown" "Navigate to Millenium Park" "Web Search Green Bean Casserole" "Open Calculator" "Opean Alarm Clock" "Launch Phone book" Personal Health Planner ... "How many days until Christmas" Travel Planner "Show airline directory" "Find hotels" "Rent a car" "Check flight status" "Currency converter" Cluzee Car Mode Access Daily Brief, Personal Radio, Search, Maps, Directions etc..

A few fragments (sub-sentences) of this incorrectly marked sentence describe the need for read calender permission ("*...My reminder ... Add reminder ...*"). However, inaccurate identification of sentence boundaries causes the underlying NLP infrastructure produces a incorrect annotation. Such incorrect annotation is propagated to subsequent phases of WHYPER, ultimately resulting in inaccurate identification of a permission sentence. Overall, as the current research in the filed of NLP advances the underlying NLP infrastructure, a significant number of false negatives will be reduced.

7.3.2 RQ2: Comparison to keyword-based searching

In this section, we answer RQ2 by comparing WHYPER to a keyword-based searching approach in identifying permission sentences. As described in Section 7.2, we manually measured the number of permission sentences in the application descriptions. Furthermore, we also manually computed the precision (P), recall (R), f-score (F_S), and accuracy (Acc) of WHYPER as well as precision (P'), recall (R'), f-score (F'_S), and accuracy (Acc') of keyword-based searching in identifying permission sentences. We then calculated the improvement in using WHYPER against keyword-based searching as $\Delta P = P - P'$, $\Delta R = R - R'$, $\Delta F_S = F_S - F'_S$, and $\Delta Acc = Acc - Acc'$. Higher values of ΔP, ΔR, ΔF_S, and ΔAcc are indicative of better performance of WHYPER against keyword-based search.

Table 7 shows the comparison of WHYPER in identifying permission sentences to keyword-based searching approach. Columns "ΔP", "ΔR", "ΔF_S", and "ΔAcc" list percentage values of increase in the precision, recall,

[10]Note that the incorrect grammar, punctuation, and spacing are a reproduction of the original description.

Table 7: Comparison with keyword-based search

Permission	ΔP%	ΔR%	ΔF_S%	ΔAcc%
READ_CONTACTS	50.4	1.3	31.2	7.3
READ_CALENDAR	39.3	1.5	26.4	9.2
RECORD_AUDIO	36.9	-6.6	24.3	6.8
Average	41.6	-1.2	27.2	7.7

f-scores, and accuracy respectively. Our results show that, in comparison to keyword-based searching, WHYPER effectively identifies permission sentences with the average increase in precision, F-score, and accuracy of 41.6%, 27.2%, and 7.7% respectively. We indeed observed a decrease in average recall by 1.2%, primarily due to poor performance of WHYPER for RECORD_AUDIO permission.

However, it is interesting to note that there is a substantial increase in precision (average 46.0%) in comparison to keyword-based searching. This increase is attributed to a large false positive rate of keyword-based searching. In particular, for descriptions related to READ_CONTACTS permission, WHYPER resulted in only 18 false positives compared to 265 false positives resulted by keyword-based search. Similarly, for descriptions related to RECORD_AUDIO, WHYPER resulted in 64 false positives while keyword-based searching produces 338 false positives.

We next present illustrative examples of how WHYPER performs better than keyword-based search in context of false positives. One major source of false positives in keyword-based search is confounding effects of certain keywords such as *contact*. Consider the sentence "*contact me if there is a bad translation or you'd like your language added!*". The sentence describes that developer is open to feedback about his application. A keyword-based searching incorrectly identifies this sentence as a permission sentence for READ_CONTACTS permission. However, the word *contact* here refers to an action rather than a resource. In contrast, WHYPER correctly identifies the word *contact* as an action applicable to pronoun *me*. Our framework thus correctly classifies the sentences as a permission-irrelevant sentence.

Consider another sentence "*To learn more, please visit our Checkmark Calendar web site: calendar.greenbeansoft.com*" as an instance of confounding effect of keywords. The sentence is incorrectly identified as a permission sentence for READ_CALENDAR permission because it contains keyword *calendar*. In contrast, WHYPER correctly identifies "Checkmark Calendar" as a named entity rather than resource *calendar*. Our framework thus correctly identifies the sentences as not a permission sentence.

Another common source of false positives in keyword-based searching is lack of semantic context around a keyword. For instance, consider the sentence "*That's what this app brings to you in addition to learning numbers!*". A keyword-based search classifies this sentence as an permission sentence because it contains the keyword *number*, which is one of the keywords for READ_CONTACTS permission as listed in Table 6. However, the sentence is actually describing the generic numbers rather than phone numbers. Similar to keyword-based search, our framework also identifies word *number* as a candidate match for subordinating resource *number* in READ_CONTACTS permission (Figure 10). However, the identified semantic action on candidate resource *number* for this sentence is *learning*. Since *learning* is not an applicable action to *phone number* resource in our semantic graphs, WHYPER correctly classifies the sentences as not a permission sentence.

The final category, where WHYPER performed better than keyword-based search, is due to the use of synonyms. For instance, *address book* is a synonym for *contact* resource. Similarly *mic* is synonym for *microphone* resource. Our framework, leverages this synonym information in identifying the resources in a sentence. Synonyms could potentially be used to augment the list of keywords in keyword-based search. However, given that keyword-based search already suffers from a very high false positive rate, we believe synonym augmentation to keywords would further worsen the problem.

We next present discussions on why WHYPER caused a decline in recall in comparison to keyword-based search. We do observe a small increase in recall for READ_CONTACTS (1.3%) and READ_CALENDAR (1.5%) permission related sentences, indicating that WHYPER performs slightly better than keyword-based search. However, WHYPER performs particularly worse in RECORD_AUDIO permission related descriptions, which results in overall decrease in the recall compared to keyword-based search.

One of the reasons for such decline in the recall is an outcome of the false negatives produced by our framework. As described in Section 7.3.1 incorrect identification of sentence boundaries and limitations of underlying NLP infrastructure caused a significant number of false negatives in WHYPER. Thus, improvement in these areas will significantly decrease the false negative rate of WHYPER and in turn, make the existing gap negligible.

Another cause of false negatives in our approach is inability to infer knowledge for some ‘resource’-*‘semantic action’* pairs, for instance, ‘microphone’-*‘blow into’*. We further observed, that with a small ***manual effort in augmenting semantic graphs*** for a permission, we could significantly bring down the false negatives of our approach. For instance, after a precursory observation of false negative sentences for RECORD_AUDIO permission manually, we augmented the semantic graphs with just two resource-*action* pairs (1. microphone-*blow into* and 2. call-*record*). We then applied WHYPER with the augmented semantic graph on READ_CONTACTS permission sentences.

The outcome increased ΔR value from -6.6% to 0.6% for RECORD_AUDIO permission and an average increase of 1.1% in ΔR across all three permissions, without affecting values for ΔP. We refrained from including such modifications for reporting the results in Table 7 to stay true to our proposed framework. In the future, we plan to investigate techniques to construct better semantic graphs for permissions, such as mining user comments and forums.

7.4 Summary

In summary, we demonstrate that WHYPER effectively identifies permission sentences with the average precision, recall, F-score, and accuracy of 80.1%, 78.6%, 79.3%, and 97.3% respectively. Furthermore, we also show that WHYPER performs better than keyword-based search with an average increase in precision of 40% with a relatively small decrease in average recall (1.2%). We also provide discussion that such gap in recall can be alleviated by improving the underlying NLP infrastructure and a little manual effort. We next present discussions on threats to validity.

7.5 Threats to Validity

Threats to external validity primarily include the degree to which the subject permissions used in our evaluations were representative permissions. To minimize the threat, we used permissions that guard against the resources that can be subjected to privacy and security sensitive operations. The threat can be further reduced by evaluating WHYPER on more permissions. Another threat to external validity is the representativeness of the description sentences we used in our experiments. To minimize the threat we randomly selected actual Android application descriptions from a snapshot of the meta-data of 16001 applications from Google Play store (dated January 2012).

Threats to internal validity include the correctness of our implementation in inferring mapping between natural language description sentences and application permissions. To reduce the threat, we manually inspected all the sentences annotated by our system. Furthermore, we ensured that the results were individually verified and agreed upon by at least two authors. The results of our experiments are publicly available on the project website [1].

8 Future work and Dissertation Goals

The goal of my dissertation is to improve developer / tester / end-user productivity by automatically inferring the semantic information in terms of formal specifications from the textual descriptions in software artifacts. In particular, these formal specifications will be directed towards augmenting existing tools/frameworks that currently do not work with natural language specifications.

For this examination report, I have used a portion of my dissertation to provide two instances of the generic NL based approach for inferring specification from API documents. [48] uses generic NLP based techniques to infer parameter specifications from API documents to assist software quality assurance practitioners to better test the software using existing tools that work on formal specifications. *WHYPER* [47] further extends the NLP infrastructure to bridge the gap between user expectations and software functionality in context of mobile software. In particular, *WHYPER* finds the mapping between the permissions requested by a mobile software and the applications descriptions that state how that particular permission is going to be used. In future, I plan to improve the presented NLP infrastructure by further developing techniques to improve developer productivity. I next describe the future work that I plan to undertake to achieve my dissertation goal, followed by an tentative time-line for intermediate milestones

8.1 Temporal Constraints in API

Software reuse is commonly practiced since the advent of software development [21]. Software reuse facilitates development of larger, more complex, and timely-delivered software systems [22]. 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.

Library developers commonly describe legal usage in natural language text in API documents. Typically, such documents are provided to client-code developers through online access, or are shipped with the API code. For example, J2EE's API documentation[11] is one of the popular API documents. Typically, an API document will describe the both the constraints on the method parameters as well as the constraints in terms of the methods the must be invoked pre/post the current method.

Our previous work [48], focused on inferring the parameter constraints, in contrast the proposed work will focus on the temporal constraints of the methods in an API. A temporal relation is defined as *an interpropositional relation that communicates the simultaneity or ordering in time of events or states.* In terms of API method invocations, I interpret temporal relationships as *'the allowed sequence of invocations of methods'*.

In research literature, there are some mining approaches [5, 65, 67, 75] that partially address the problem. However, all of these approaches rely on the availability of the source code that use API methods under investigation. Furthermore, these approaches are limited by both the quantity as well as quality of data set (read source code) available. To address the limitation I propose to extend the natural language infrastructure developed previously [48, 47] to focus specifically on the temporal constraints. Furthermore, I propose to evaluate the effectiveness of proposed work on real world API's: 1) Java 7 and 2) Amazon S3 REST API. The former is generic language API, while the later is a web based API directed towards cloud-based data storage. The temporal constraints inferred for Java 7 will be compared with the existing approaches to evaluate the effectiveness of proposed work. In contrast, Amazon S3 REST API (relatively recent) representative of the class of API's where neither source code of API nor enough usage examples are available.

In summary, the proposed work will leverage natural language description of API's to infer temporal constraints of method invocations. As the proposed work analyzes API documents in natural language, it can be reused independent of the programming language of the library. The proposed work in general, will make the following major contributions:

- A technique that effectively infers temporal constraints of method invocations.
- A prototype implementation of our approach based on extending the Stanford Parser [39, 57], which is a natural language parser to derive the grammatical structure of sentences. An open source implementation of the prototype will be made available publicly.
- An evaluation of proposed approach on Java 7 and Amazon S3 REST API

8.2 Method Mapping across API

With ever increasing computing platforms finding there way to consumers, software developers increasingly release different versions of their application either to address a business requirement or to survive in competing market. For example, a mobile software developers often release their applications for all the popular mobile platforms such as Android, iOS, and Windows. In context of desktop software, many well-known projects such as JUnit, Hibernate provide multiple versions in different languages, in an attempt to lure developer community to use these libraries across different languages.

Manually migrating a software from one platform(/language) to another is prohibitively time consuming and may be error prone. Recent researches [26, 74, 65, 73] partially alleviate the issue. As they require a programmer to manually describe how API's of a source platform(/language) maps to API's of the target platform(/language). Given a typical platform(/language) expose a large number of API's for developers to reuse, the aforementioned researches and tools typically support a limited subset of API's.

In the literature, there exists approaches that address the problem finding mapping between API's, leveraging static [74, 65] and dynamic [26] analysis. However, these approach rely existence of manually ported (or at least functionally similar) software across source and target API's. However, for two arbitrary libraries, there may not be implementations of the same project in these two libraries, and the APIs used in such projects may be limited. What

[11] `http://download.oracle.com/javaee/1.6/api/`

Table 8: Milestones in achieving my dissertation goals

ID	TASK	Start	Finish	Duration	Fall'13	Spring'14	Summer'14
1	Temporal Constraints in API	08/15/13	12/31/13	139	■		
2	Inferring Method Mappings Across API	01/01/13	05/15/14	135		■	
3	Dissertation Writing	05/06/14	08/14/14	92			■

if such mapping software pairs are not available. Moreover, does existence of such pair guarantee coverage of all API elements?

To address these shortcomings I propose a novel approach for inferring likely mapping between API's. In particular, I propose the use of Natural Language processing of API documents to infer such mapping. I plan to leverage these method descriptions along with programming information such as types of parameters variables, return variables, and thrown exceptions to find the likely mapping. It is not trivial to use natural language processing on API documents to infer likely mappings. However, I am motivated by our intuition that :"API documents are written for developers, thus there should be a significant overlap in the style/language used in these documents". We plan to leverage this similarity of style to infer the likely mappings.

In summary, the proposed work leverages natural language description of API's to infer likely mapping thus facilitating cross API migration of applications. As the proposed work analyzes API documents in natural language, it can be reused independent of the programming language of the library. The proposed work will makes the following major contributions:

- A technique that effectively infers mapping across source and target API.
- A prototype implementation of our approach based on extending the Stanford Parser [57], which is a natural language parser to derive the grammatical structure of sentences. An open source implementation of the prototype will be made available publicly.
- An evaluation of proposed approach on J2ME and Android API

8.3 Time-line for milestones

Table 8 shows the intermediate milestones and my planned schedule for achieving those milestones. Column "ID" lists the identifier of the future task, Column "Task" lists the name of the task. Columns "Start" and "Finish" lists the starting and finishing date respectively. Column "Duration" list the duration of task in number of days. Columns "Fall'13","Spring'14", and "Summer'14" are the current/future academic semester and a cell shaded black under these columns represents that task will be active in that semester.

Along with these proposed milestones, I plan to submit my findings on temporal constraints in API to ASE 2014 (submission deadline around April 2014) and my findings on inferring Method Mappings Across API to ICSE 2015(submission deadline around August 2014).

9 Conclusion

A lot of specifications pertaining to software are in the form of Natural Language (NL) text distributed across various software artifacts such as requirements documents, design documents, application descriptions and API documents. Clearly the ambiguity accompanied with NL hinders tool-based verification. To address this issue, this report presents a natural language processing framework to automate the task of inferring formal specifications from NL software artifacts. Specifically, in this report, I presented two recent research efforts that I have conducted in developing/applying NLP techniques for discovering specifications from NL software artifacts. Our evaluation results show great promise in applying NLP techniques to assist testers/end-users in achieving their goals. Furthermore, I also outline my dissertation plan along with intermediate deliverables to achieve my goal.

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] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proc. CASSIS, LNCS vol. 3362*, 2004.

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

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

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

[7] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *Proc. 6th WiSec*, 2013.

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

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

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

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

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

[13] 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.

[14] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proc. of 18th ISOC NDSS*, 2011.

[15] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of 9th USENIX OSDI*, pages 1–6, 2010.

[16] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proc. 20th USENIX Security Symposium*, page 21, 2011.

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

[18] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A Survey of Mobile Malware in the Wild. In *Proc. ACM Workshop on SPSM)*, 2011.

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

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

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

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

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

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

[25] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proc. of 5th TRUST*, pages 291–307, 2012.

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

[27] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and accurate zero-day Android malware detection. In *Proc. of 10th MobiSys*, pages 281–294, 2012.

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

[29] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46-1, pages 317–330, 2011.

[30] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46-6:62–73, 2011.

[31] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *ACM SIGPLAN Notices*, volume 46-6, pages 50–61, 2011.

[32] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, volume 46-6, pages 317–328, 2011.

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

[34] 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.

[35] 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.

[36] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the Droids you're looking for: Retrofitting Android to protect data from imperious applications. In *Proc. 18th ACM CCS*, pages 639–652, 2011.

[37] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *ACM Sigplan Notices*, volume 45-10, pages 36–46, 2010.

[38] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proc. 32nd ICSE*, volume 1, pages 215–224, 2010.

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

[40] 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.

[41] 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.

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

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

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

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

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

[47] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: towards automating risk assessment of mobile applications. In *Proc. 22nd USENIX conference on Security*, pages 527–542, 2013.

[48] 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.

[49] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proc. 19th CCS*, 2012.

[50] 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.

[51] 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.

[52] P. Robillard. Schematic pseudocode for program constructs and its computer automation by schemacode. *Communications of the ACM*, 29(11):1072–1089, 1986.

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

[54] 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.

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

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

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

[58] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proc. of 19th ICPC*, pages 71–80, 2011.

[59] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *ACM Sigplan Notices*, volume 45-1, pages 313–326, 2010.

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

[61] A. Taly, S. Gulwani, and A. Tiwari. Synthesizing switching logic using constraint solving. *International journal on software tools for technology transfer*, 13-6:519–535, 2011.

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

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

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

[65] 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.

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

[67] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proc. 10th Working Conference on MSR*, pages 319–328, 2013.

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

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

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

[71] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proc. of 21st USENIX Security Symposium*, page 29, 2012.

[72] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *Proc. 26th ASE*, pages 63–72, 2011.

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

[74] 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.

[75] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending API usage patterns. In *Pro. 23rd ECOOP*, pages 318–343, 2009.

[76] 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.

[77] 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.

[78] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. of IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

[79] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proc. of 19th NDSS*, 2012.