ABSTRACT

PANDITA, RAHUL. Inferring Semantic Information from Natural-Language Software Artifacts. (Under the direction of Laurie Williams.)

Specifications play and important role in software engineering for ensuring software quality. Not only do the specifications guide the development process by outlining what/how to reuse, they also help in the verification process by allowing testers to test the expected outcome. For instance, static and dynamic program analysis tools use formal specifications such as code contracts or assertions to detect violations of these specifications as defects. Likewise, API migration tools use predefined mapping specifications to perform automated migration. While highly desirable, oftentimes such formal specifications are missing from existing software. In contrast, these specifications are described in natural language in various software artifacts. Particularly, Application Programming Interface (API) documents that are targeted towards developers, are invaluable source of information regarding code-level specifications. However, (most of) existing developer productivity tools/frameworks are not designed to process the natural-language descriptions in software artifacts.

The goal of this work is to improve developer / tester / end-user productivity by accurately identifying specification sentences from the natural language text in software artifacts and representing them formally through adaptation of existing text analysis techniques.

Since natural-language software artifacts are often verbose, manually writing formal specifications from software artifacts may be resource intensive and error prone. To address this issue, this dissertation presents a text mining and natural language processing (NLP) framework to automate the task of inferring semantic information from natural-language in software artifacts to bridge the disconnect between the inputs required by software engineering tools/frameworks and the specifications described in natural-language software artifacts. Specifically, this dissertation reports on the process and effectiveness of applying text mining and NLP techniques in the following developer problems:

1. *Specification of a method arguments and return values* describes how to use a particular method within a class in terms of the expectations of the method arguments (preconditions) and expected return values (post-conditions). We propose a novel approach to infer these specifications from natural-language text of API documents. Our evaluation results show that our approach achieves an average of 92% precision and 93% recall

in identifying sentences that describe such specifications 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 specification sentences.

2. *Temporal specifications of an API* are the allowed sequences of method invocations in the API. We propose ICON: an approach based on machine learning and NLP for identifying and inferring formal temporal constraints to assist tool based verification. Our evaluation results indicate that ICON is effective in identifying temporal constraint sentences (from over 4000 human-annotated API sentences) with the average precision, recall, and F-score of 79.0%, 60.0%, and 65.0%, respectively. Furthermore, our evaluation also demonstrates that ICON achieves an accuracy of 70% in inferring and formalizing 77 temporal constraints from these temporal constraint sentences.

3. *API mappings* facilitate tool based language migration. We propose TMAP: **T**ext **M**ining based approach to discover likely **API** mappings using the similarity in the textual description of the source and target API documents. We evaluated TMAP by comparing the discovered mappings with state-of-the-art source code analysis based approaches: Rosetta and StaMiner. Our results indicate that TMAP on an average found relevant mappings for 57% more methods compared to previous approaches. Furthermore, our results also indicate that TMAP found, on average, exact mappings for 6.5 more methods per class as compared to previous approaches.

4. Keeping malware out of mobile application markets is an ongoing challenge. To assist third-party testers for *assessing risk of mobile applications*, we present WHYPER, a framework using NLP techniques to identify sentences that describe the need for a given permission in an application description. WHYPER achieves an average precision of 82.8%, and an average recall of 81.5% for three permissions (address book, calendar, and record audio) that protect frequently-used security and privacy sensitive resources. These results demonstrate great promise in using NLP techniques in aiding the risk assessment of mobile applications.

Inferring Semantic Information from Natural-Language Software Artifacts

by
Rahul Pandita

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2015

APPROVED BY:

William Enck	Emerson Murphy-Hill
Munindar P. Singh	Laurie Williams Chair of Advisory Committee

DEDICATION

"Would you tell me, please, which way I ought to go from here?"

"That depends a good deal on where you want to get to."

"I don't much care where "

"Then it doesn't matter which way you go."

– Lewis Carroll, *Alice in Wonderland*

To my parents; Ravi Prakash Pandita and Chandra Pandita.
For all the freedom to explore.

BIOGRAPHY

Rahul Pandita was born in Srinagar, that is the capital of the beautiful valley of Kashmir in India. He completed his Master of Science in Computer Science from North Carolina State University in 2011 and Bachelor of Engineering in Computer Science from Pune University in 2007. His research interests are in software engineering in general, with a primary goal to apply natural language processing techniques and tools to improve developer/tester/end-user productivity. During his PhD program, his internship experiences included US FDA, Silver Spring, MD, USA (Summer 2010, Fall 2011, Summer 2012); IBM India Research Labs, Bangalore, India (Summer 2011); and Accenture Technology Labs, San Jose, CA, USA (Summer 2013). He is a recipient of a Federal Group Recognition Award at FDA *for exceptional innovating, planning and execution of strategic scientific initiative towards building a semantic text mining search environment at FDA and for promoting its use in clinical adverse event management* in 2012. During his PhD program, Rahul contributed to various open source projects, inducing ICON, Whyper, BV&LC Coverage, and TMAP, with a common theme of improving developer/tester/end-user productivity. Research associated with these tools have been published (or are under review) in various Software Engineering related conferences.

ACKNOWLEDGEMENTS

This journey would not have been possible without the support of a lot of people.

First, I would like to thank my adviser Dr. Laurie Williams for supporting me through the PhD process. Laurie provided me with the freedom to explore and pursue my research directions and was very patient with me during the process. Throughout the way, Laurie provided me with valuable technical and professional advice, helping me to grow as a researcher. I am also thankful to Dr. Tao Xie for introducing me to the wonderful world of software engineering research and shaping me as a professional during my initial years in the doctoral program. I know I will continue to benefit from my interactions with them throughout my career.

I would like to extend my gratitude towards my committee members: Dr. William Enck, Dr. Emerson Murphy-Hill, and Dr. Munindar Singh. First, I thank them for serving as my committee members and providing valuable feedback on my work. In addition, I have been very lucky to work closely with each of my committee members. These work opportunities have helped me a lot professionally and made me appreciate the diverse research areas and approaches.

I wish to acknowledge the support of my research collaborators: Dr. Raoul Jetley and Dr. Sithu Sudarsan at ABB Research India (formerly at United States Food and Drug Administration) to introduce me to the text mining concepts. I would also like to thank my mentor at IBM India Research Labs Dr. Saurabh Sinha for helping me understand the concepts of program analysis. I am thankful to Dr. Teresa Tung from Accenture Research Labs for allowing me to extend my university research into a corporate study. I am grateful to Nikolai Tillmann and Peli de Halleux at Microsoft Research for their support on Pex and their guidance on the testing infrastructure. I am also thankful to the reviewers of various conferences for providing the valuable feedback on the submitted drafts.

The members of Realsearch group have been a source of constant support. I would like to take this opportunity to acknowledge the help extended by the members for: countless reviews of the half baked drafts, providing feedback on the presentations, critiquing the study design, and on occasions helping out with the studies as well. I am also thankful for the help extended by members of Automated Software Engineering Group at North Carolina State University (now at University of Illinois at Urbana-Champaign).

Heartfelt thanks go to all the helpful staff at the Department of Computer Science at the North Carolina State University. Special thanks go to my dearest friends Suresh Thum-

malapenta, Kunal Taneja, Jeehyun Hwang, Xusheng Xiao, Da Young Lee, Padmashree Ravindra, Niranjan "Kulla" Kulkarni, Suprit "Pats" Patankar, John Majikes, John Slankas, Patrick Morrison, Brittney Johnson, and Yoonki Song for making Raleigh a home away home for me.

The research presented in this dissertation has been supported by (in part) by USA National Security Agency (NSA) Science of Security Lablet at NCSU; NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603, CNS-1222680, CNS-1253346, CNS-0720641, and CCF-0725190; ARO grant W911NF-08-1-0443 and ARO grant W911NF-08-1-0105 managed by NCSU SOSI; and an NCSU CACC grant. Portions of this dissertation were previously published at ICSE'12, USENIX'13. The material included in this dissertation has been updated and extended.

I am very thankful to my family Chandra Pandita (Mom), Ravi Parkash Pandita (Dad), and Rohit Pandita (Brother) who were my constant source of support and encouragement throughout my journey as a doctoral student. Last but not the least I would like to thank my wife, Neha Sinha, for believing in me and for providing me the kind of support that I needed while writing this dissertation.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **ix**

LIST OF FIGURES . . . . . **x**

Chapter 1 Introduction **1**
1.1 Motivation 1
1.2 Summary 2
1.3 Challenges 4
1.4 Scope 5
1.5 Contributions 6
1.6 Outline 7

Chapter 2 Background **8**
2.1 Natural Language Processing 8
2.2 Text Mining 10
2.3 Formal representation used in this work 13

Chapter 3 Inferring Method Specifications from Natural Language API Descriptions **15**
3.1 Introduction 15
3.2 Motivating Examples 18
3.3 Approach 20
3.3.1 Parser 20
3.3.2 Pre-Processor 21
3.3.3 Text Analysis Engine 23
3.3.4 Post-processor 26
3.3.5 Code Contract Generator 28
3.4 Evaluation 29
3.4.1 Subjects 30
3.4.2 Evaluation Results 31
3.4.3 Threats to Validity 36
3.5 Discussion and Future work 36
3.5.1 Limitations: 37
3.6 Chapter Summary 38

Chapter 4 ICON: Inferring Temporal Constraints from Natural Language API Descriptions **39**
4.1 Introduction 39
4.2 Motivating Example 42

4.3 ICON Overview . . . 43
4.3.1 Preprocessor . . . 43
4.3.2 NLP Parser . . . 45
4.3.3 Candidate Identifier . . . 45
4.3.4 Text Analysis Engine . . . 48
4.3.5 Constraint Extractor . . . 51
4.3.6 Semantic-Graph Generator . . . 53
4.3.7 Type Analysis . . . 53
4.4 Evaluation . . . 54
4.4.1 Subjects . . . 55
4.4.2 Evaluation Setup . . . 56
4.4.3 Results . . . 58
4.4.4 Summary . . . 61
4.4.5 Threats to Validity . . . 61
4.5 Limitations and Future work . . . 61
4.6 Chapter Summary . . . 62

Chapter 5 WHYPER: Towards Automating Risk Assessment of Mobile Applications 63
5.1 Introduction . . . 63
5.2 Android Applications . . . 66
5.3 WHYPER Overview . . . 66
5.3.1 Use Cases and Threat Model . . . 67
5.4 WHYPER Design . . . 69
5.4.1 Preprocessor . . . 69
5.4.2 NLP Parser . . . 72
5.4.3 Intermediate-Representation Generator . . . 73
5.4.4 Semantic Engine (SE) . . . 76
5.4.5 Semantic-Graph Generator . . . 77
5.5 Evaluation . . . 79
5.5.1 Subjects . . . 80
5.5.2 Evaluation Setup . . . 80
5.5.3 Results . . . 82
5.5.4 Summary . . . 88
5.5.5 Threats to Validity . . . 88
5.6 Discussions and Future work . . . 89
5.7 Chapter Summary . . . 90

Chapter 6 Discovering Likely Mappings Between APIs using Text Mining . . . 92
6.1 Introduction . . . 92
6.2 Example . . . 95
6.3 Approach . . . 97

6.3.1 Indexer . . . 98
6.3.2 Query Builder . . . 101
6.3.3 Searcher . . . 103
6.3.4 Implementation . . . 104
6.4 Evaluation . . . 104
6.4.1 Subjects . . . 105
6.4.2 Evaluation Setup . . . 106
6.4.3 Results . . . 108
6.5 Limitation and Future work . . . 113
6.6 Chapter Summary . . . 114

Chapter 7 Related Work . . . 115
7.1 Formal Specifications . . . 115
7.2 NLP in Software Engineering . . . 116
7.3 Program Comprehension . . . 117
7.4 Program Synthesis . . . 118
7.5 Language Migration . . . 119
7.6 Information Retrieval . . . 121
7.7 Query Formulation . . . 121

Chapter 8 Future Work . . . 122
8.1 Generic Limitations . . . 122
8.2 Coding-specific activities . . . 124
8.3 Self Assurance . . . 125
8.4 WHYPER : Generalization to Other Permissions . . . 125
8.5 Long term goal . . . 126

Chapter 9 Summary and Lessons Learned . . . 127
9.1 Summary . . . 127
9.2 Lessons learned . . . 129

REFERENCES . . . 131

LIST OF TABLES

Table 3.1 Categories of Shallow Parsing Semantic Templates. 25
Table 3.2 Statistics of Subject classes and Evaluation results 33

Table 4.1 Semantic Templates 50
Table 4.2 Evaluation Results (Identification) 58
Table 4.3 Evaluation Results (Inference) 59

Table 5.1 Examples of Semantic Templates for Stanford Typed Dependencies 75
Table 5.2 Statistics of Subject permissions 81
Table 5.3 Evaluation results 82
Table 5.4 Keywords for Permissions 86
Table 5.5 Comparison with keyword-based search 86

Table 6.1 Query Results 97
Table 6.2 Evaluation Results 109
Table 6.3 Comparison with Sniff 112

LIST OF FIGURES

Figure 2.1 An example of POS tags and Chunking annotation. 9
Figure 2.2 An example of Stanford Typed Dependencies representation. 11
Figure 2.3 An example of Intermediate Representation. 14

Figure 3.1 Description of a question posed in a Java forum 19
Figure 3.2 Overview of our approach to infer method specification 21
Figure 3.3 The method description of the `DefineObjectProperty` method 22
Figure 3.4 Specifications in format of FOL expressions extracted by our NLP Parser . 24
Figure 3.5 FOL expression after synonym analysis and compaction 29
Figure 3.6 Example of the inferred specifications for `DefineObjectProperty` method 30

Figure 4.1 Query posted on StackOverflow forum about Amazon S3 REST API . . . 42
Figure 4.2 Overview of the ICON approach 44
Figure 4.3 Sentence annotated with Stanford dependencies 46
Figure 4.4 An example FOL-like representation of a sentence 51
Figure 4.5 Semantic Graph for the `Object` method in Amazon S3 REST API 54

Figure 5.1 Overview of WHYPER 68
Figure 5.2 Overview of WHYPER framework 70
Figure 5.3 Sentence annotated with Stanford dependencies 73
Figure 5.4 First-order logic representation of annotated sentence in Figure 5.3 73
Figure 5.5 Semantic Graph for the READ_CONTACT permission 78

Figure 6.1 `drawString` API method description from `Graphics` class in the Java ME API 95
Figure 6.2 `drawText` API method description from `Canvas` class in the Android API 96
Figure 6.3 Overview of ICON approach 98
Figure 6.4 Description of `Iterator.hasNext` method from Java API 102
Figure 6.5 An example of query generated by TMAP 102

CHAPTER 1

Introduction

1.1 Motivation

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. An example of such specification is 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.

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 regarding 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, J2EEs API documentation is one of the popular API documents.

API documentation provides developers with useful information about class/interface hierarchies within a software. Additionally, API documents also provide 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 invaluable 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-conditions or the post conditions are violated. However, such tools are not designed to work with the natural language descriptions, and often require more formal specifications. Thus, a disconnect exists between the inputs required by these tools and the natural language specifications described in software artifacts.

The goal of this work is to improve developer / tester / end-user productivity by accurately identifying specification sentences from the unconstrained natural language text in software artifacts and representing them formally through adaptation of existing text analysis techniques.

The formal representation of the specification sentences are meant to bridge the gap between the existing tools that help the target audience and inputs required by tools. This dissertation presents my recent research projects for developing/applying text mining and Natural Language Processing (NLP) techniques to discover specifications from natural language software artifacts. The evaluation of these efforts show promising results to improve developer/tester productivity. For example, a formal representation of API method expectations could assist a developer in writing better code. Similarly, a formal representation of appropriate usage of an API method could be beneficial to a quality assurance practitioner by explicitly delineating the illegal usages of an API. From an end users perceptive automated analysis of application description can empower them by answering a variety of questions regarding privacy and security such as: how a users private data will be used.

1.2 Summary

This section summarizes the individual instances of inferring formal specifications form natural language artifacts for improving developer, tester, and end user productivity.

1. **Method arguments and return value specifications**: Specification of a method arguments and return values describe how to use a particular method within a class in terms of the expectations of the method arguments (pre-conditions) and expected return values (post-conditions). In practice, most libraries do not come with formal specifications, thus hindering tool-based verification. To address this issue, we propose a novel approach to infer formal specifications from natural language text of API documents. Our evaluation results show that our approach achieves an average of 92% precision and 93% recall

in identifying sentences that describe method arguments and return value specifications 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 method arguments and return value specification sentences.

2. **Temporal specifications for method invocation** Temporal specifications of an Application Programming Interface (API) are the allowed sequences of method invocations in the API. We propose ICON: an approach based on Machine Learning (ML) and Natural Language Processing (NLP) for identifying and inferring formal temporal constraints to assist tool based verification. To evaluate our approach, we apply ICON to infer and formalize temporal constraints from the `Amazon S3 REST` API, the `PayPal Payment REST` API and the `java.io` package in the JDK API. Our results indicate that ICON is effective in identifying temporal constraint sentences (from over 4000 human-annotated API sentences) with the average precision, recall, and F-score of 79.0%, 60.0%, and 65.0%, respectively. Furthermore, our evaluation also demonstrates that ICON achieves an accuracy of 70% in inferring and formalizing 77 temporal constraints from these temporal constraint sentences.

3. **API mappings for language migration** API mappings facilitate tool based language migration. Previously, researchers proposed to learn mapping specifications by mining existing code-bases. However, these approaches require as input a manually ported (or at least functionally similar) code across source and target API's. To address the shortcoming, we propose TMAP: ***T**ext **M**ining based approach to discover likely **API** mappings* using the similarity in the textual description of the source and target API documents. To evaluate our approach, we apply TMAP to discover API mappings for 15 classes across: 1) `Java` and `C#` API; 2) `Java ME` and `Android` API. We next compare the discovered mappings with state-of-the-art source code analysis based approaches: Rosetta and StaMiner. Our results indicate that TMAP on an average found relevant mappings for 57% more methods compared to previous approaches. Furthermore, our results also indicate that TMAP found, on average, exact mappings for 6.5 more methods per class with a maximum of 21 additional exact mappings for a single class as compared to previous approaches.

4. **Risk assessment of mobile applications** Application markets such as Apple's App Store and Google's Play Store have played an important role in the popularity of smartphones

and mobile devices. However, keeping malware out of application markets is an ongoing challenge. To assist third-party quality assurance practitioners in assessing risk, this study presents the first step by automatically inferring *expected functionality* from the natural language application description. Specifically, we focus on permissions for a given application and examine whether the application description provides any indication for why the application needs a permission. We present WHYPER, a framework using Natural Language Processing (NLP) techniques to identify sentences that describe the need for a given permission in an application description. WHYPER achieves an average precision of 82.8%, and an average recall of 81.5% for three permissions (address book, calendar, and record audio) that protect frequently-used security and privacy sensitive resources. These results demonstrate great promise in using NLP techniques in aiding the risk assessment of mobile applications.

1.3 Challenges

In this section we briefly summarize some of the technical challenges addressed by this dissertation in applying text mining and natural language processing techniques on natural language software engineering artifacts.

1. *Ambiguity*. Existing linguistic analysis techniques focus on well-written documents such as news articles [SNL99]. In contrast, method descriptions are often not well-written. For instance, consider the sentence from the `File` class in the C# .NET Framework: *"true if path is an absolute path; otherwise false"*. This sentence does not have the main subject and the verb, which may cause the linguistic techniques to behave undesirably.

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

3. *Semantic Equivalence*. A legal usage in natural language can be described in different words and semantic structures. For instance, consider the following two fragments that describe the same specification: *"name can contain numbers, underscores..."* and *"name consists of numbers and/or underscores..."*. Thus, semantic equivalence of legal usage described in different ways needs to be identified. The proposed text mining and natural language processing techniques must be reasonably resilient to such semantic equivalence.

4. *Confounding Effects*. Certain keywords have confounding effects and are important depending on the context. For example, to find sentences related to "user contacts" in an application description keyword "contact" exhibits a confounding effect. For instance, *'... displays user contacts, ...'* vs *'... contact me at abc@xyz.com'*. The first sentence fragment refers to a sensitive operation while the second fragment does not. However, both fragments include the keyword "contact". The proposed techniques must consider the context of a sentence into account for such cases.

5. *Semantic Inference*. Natural language is very expressive, and oftentimes the sentences implicitly convey information about concepts without explicitly referring to them. For example, consider the problem to identify sentences that describe a sensitive operation such as *reading user contacts*. Oftentimes, sentences do not even explicitly contain the keyword "contact". For instance, *"share... with your friends via email, sms"*. The sentence fragment describes the *need to read contacts*; however the "contact" keyword is not used. The proposed text mining and natural language processing techniques should be able to perform such semantic reasoning.

1.4 Scope

The approaches presented in this dissertation focuses on inferring semantic information from API documents with the exception of WHYPER that works on mobile application descriptions. However, the approaches presented in the dissertation addresses the generic issues of applying NLP techniques on software artifacts and thus may be extended to other software artifacts as well.

Our approach to infer method specifications [PXZ+12] infers method specifications in terms of pre- and post- condition on method parameters. To evaluate our approach we used

API method descriptions written in the C# langauge of .NET platform. ICON: our approach for inferring method sequence specifications has been evaluated on the API written for Java language as well as language agnostic REST APIs. TMAP: our approach to infer method mappings across API has been evaluated on the following language/platform pairs: 1) Java and C#; and 2) Android and J2ME. The techniques proposed in these approaches are programming language agnostic and can be applied to various API's with minor changes to the pre-processing infrastructure. The pre-processing infrastructure is responsible to extract the desired descriptive text from an natural language artifacts.

Finally, WHYPER [PXY+13, YXP+14] approach has been implemented for application descriptions in Android ecosystem. However, the WHYPER infrastructure can be leveraged to perform similar analysis in other mobile application ecosystems such as the Windows Phone and the iOS.

The presented techniques were implemented using the "Stanford Parser" [KM03a] as our choice of external NLP annotation library. However, this dissertation uses the Stanford Parser as a black box implementation of the NLP functionality to perform analysis thus making the proposed techniques realizable through NLP annotation libraries other than Stanford Parser. As the accuracy of the external NLP annotation libraries improves accuracy of the approaches proposed in this dissertation increases and some of the preprocessing steps such as annotation of programming keywords may be rendered redundant. Finally, this dissertation does not directly contributes to the techniques in the fields of text mining and machine learning. However, this dissertation proposes enhancements to the techniques in these fields for addressing problems in software engineering domain.

1.5 Contributions

This section summarizes the main contributions of this dissertation.

- An approach that effectively identifies natural language sentences (in the API documents for a library) that describe method parameter and return value specifications, and a technique to infer specifications from the identified specification sentences.
- An ML- and NLP-based approach that effectively infers formal temporal constraints of method invocations.
  - All our experimental subjects, results, and artifacts are publicly available online. [proa].

- An approach that uses NLP techniques to help bridge the semantic gap between what mobile applications do and what users expect them to do.
  - A publicly available prototype implementation of our approach is available online [proc].
- A text mining based approach that effectively discovers mapping between source and target API.
  - A prototype implementation of our approach based on extending the Apache Lucene [luc]. An open source implementation of our prototype is publicly available online. [prob]
  - The evaluation results and artifacts are publicly available online. [prob]

1.6 Outline

The remainder of this dissertation as structured as follows: Chapter 2 provides a brief background on text mining. Chapter 3 describes our approach to infer method specifications from natural language descriptions API documents. Chapter 4 introduces our approach to infer temporal constraints method descriptions of API documents. Chapter 5 describes our approach to automatically identify the permission describing sentences in a mobile applications. Chapter 6 presents our work to infer similar methods between different API's using text mining. Chapter 7 discusses the related work. Chapter 8 describes the future-work. Finally, Chapter 9 summarizes the contributions of this dissertation and discuss the important lessons learned during the course of this dissertation.

CHAPTER 2

Background

This chapter briefly introduces the concepts and techniques from natural language processing, information retrieval, and text mining domain that have been used/adapted in this work. The remainder of the chapter is organized as follows. Section 2.1 introduces the concepts from natural language processing used in this dissertation. Section 2.2 briefly introduces text mining. Finally, Section 2.3 describes formal representation used in this work.

2.1 Natural Language Processing

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 [dMMM06, dMM08, KM03b, KM03c] 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** [KM03b, KM03c]. Also known as *'word tagging'* and *'grammatical tagging'*, 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

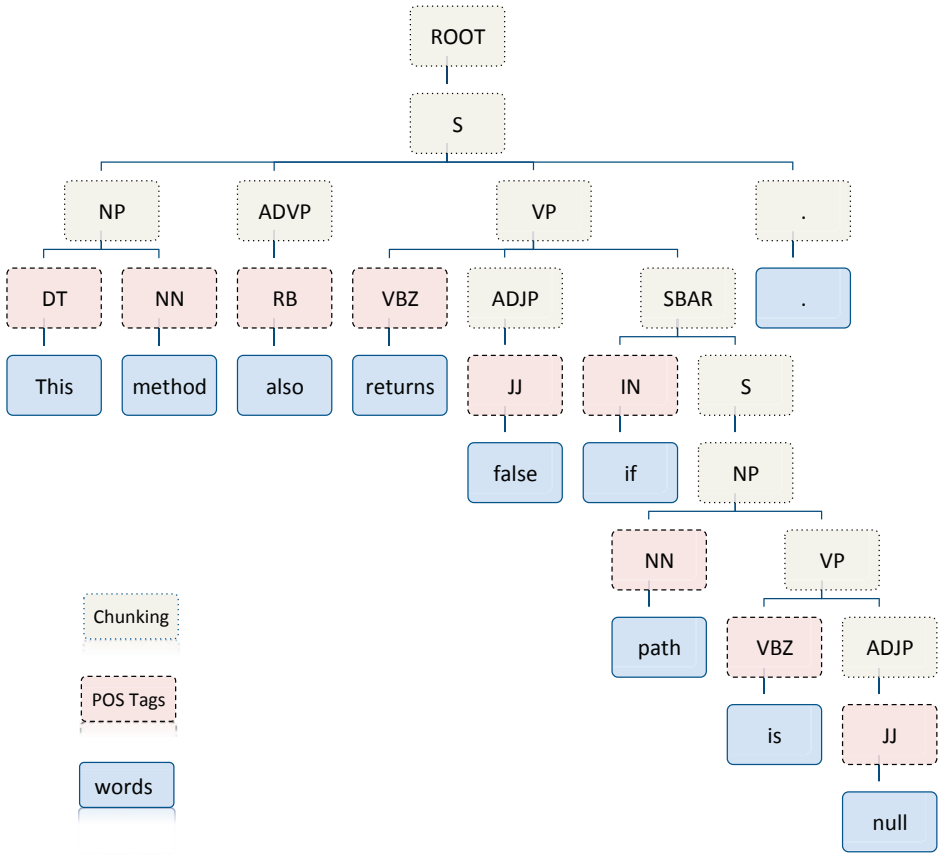

Figure 2.1: An example of POS tags and Chunking annotation for comment "This method also returns false if path is null".

the art approaches been have demonstrated to achieve 97% [SNL99] accuracy in classifying POS tags for well written news articles. Figure 2.1 shows the a sentence annotated with POS tags.

- **Chunking**. Also known as phrase and clause parsing, 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% [SNL99] accuracy in classifying phrases and clauses over well written news articles. Figure 2.1 shows the a sentence annotated with chunking tags.

- **Typed Dependencies** [dMMM06, dMM08]. 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. Figure 2.1 shows the a sentence annotated with Stanford typed dependencies.

- **Named Entity Recognition** [FGM05]. Also known as *'entity identification'* and *'entity extraction'*, these techniques are a subtask of information extraction that aims to classify words in a sentence into predefined categories such as names, quantities, expression of times, etc. These techniques help in associating predefined semantic meaning to a word or a group of words (phrase), thus facilitating semantic processing of named entities.

- **Co-reference Resolution** [RLR$^+$10, LPC$^+$11]. 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.

2.2 Text Mining

Text mining is a broad research area, including but not limited to the techniques facilitating retrieval/manipulation of useful information from a large corpus of text. A recent study [R$^+$11]

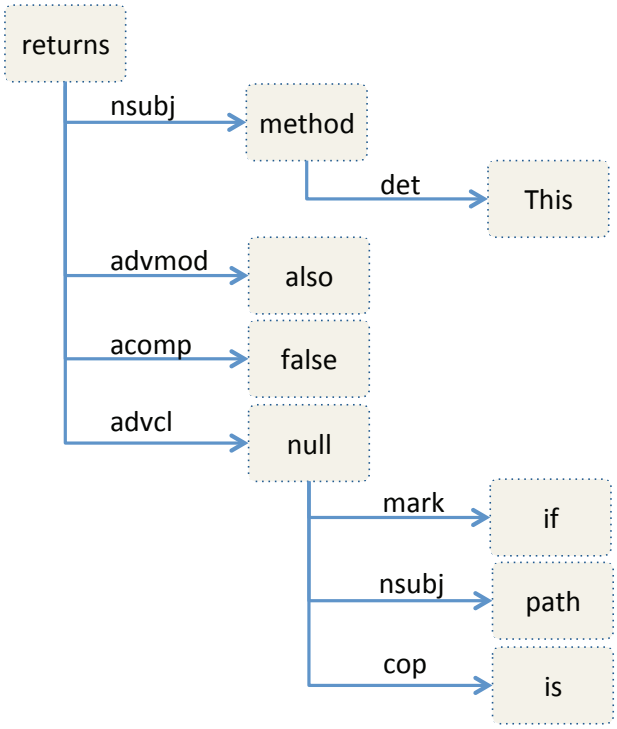

Figure 2.2: An example of Stanford Typed Dependencies for comment "This method also returns false if path is null".

indicates that a significant (35%) of business information is captured in unstructured data. A large part of this information is in documents containing narrative text, such as reports, presentations, web data, and design documents. Text mining is used to infer the knowledge in these documents through a combination of extraction and analytical methods.

As opposed to traditional data mining, text mining analyzes free-form text spread across documents (rather than data localized and maintained within a database). To analyze this data, text mining uses concepts from traditional data analytics, natural language processing, and data modeling. We next introduce the concepts from text mining that have been used in the presented approach.

Indexing [Fra92, MRS08]: Indexing is the process of extracting text data from source documents and storing them in well-defined indexes. The indexing process starts with, division of document into its constituent units, known as tokens, based on a well-defined criterion. A token is usually an individually identifiable unit of the document (such as a word) and relates to the individual terms stored in an index. Once the tokens within each document are identified, they are added to the index, with the corresponding link to the document and associated term frequencies. Term frequency is the simple count of the occurrence of a term in a document. An optional pre-processing step further assists with indexing, such as removing stop-words, grouping similar words together.

Among various indexing strategies, the use of *inverted file indexing* [Fra92] is well suited for large document collections. In the simplest form, inverted file index maintains the mapping of terms and their location in a text collection. For instance, a text document can be thought of as a collection of m words. Typically a document is made up of a sequence of n unique words such that $n <= m$. The number n is usually far less than m as most of the words are repeated while forming a document. For instance, the word "the" is repeated many times in this dissertation. The set of unique words within an index forms the "Term List" v of the index. If a pointer (say numeric location) is associated with each word in v to the location of that word in text document, the resultant data structure is a form of inverted file index. As the document collection grows, the number of documents matching a word in the index becomes sparser.

The index is oftentimes annotated with the information regarding the frequency of occurrence of each term in the document. The representation of a document as a vector of frequencies of terms is referred to as *vector space model* [Sin01, Fra92].

TF-IDF [MRS08]: Term -frequency inverse-document-frequency (`tf-idf`) is a numerical statistic that is intended to capture the importance of a term to a document in a corpus. Of-

ten used as a weighting metric in information retrieval and text mining, the `tf-idf` weight increases proportionally to frequency of the occurrence of a term in a document, however the weight is also offset by the frequency of occurrence the term in the corpus.

Cosine Similarity [Sin01]: In mathematics, cosine similarity is a numerical statistic to measure the similarity between two vectors. cosine similarity is defined as a dot product of magnitude of two vectors. In context of text mining, the cosine similarity is used to capture the similarity between two documents represented as term frequency vectors.

2.3 Formal representation used in this work

We use First-Order Logic (FOL) like 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 (or entities) and functions (or predicates). 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 $\exists vE$ and $\forall vE$ are valid expressions.

Figure 2.3 shows the graphical FOL expression. Each internal node (dotted border) represents a predicate and the children of these nodes represent the terms to that predicate. The expression is parsed as in-order traversal of tree. For instance, '*path-is-null*' form a tuple where,

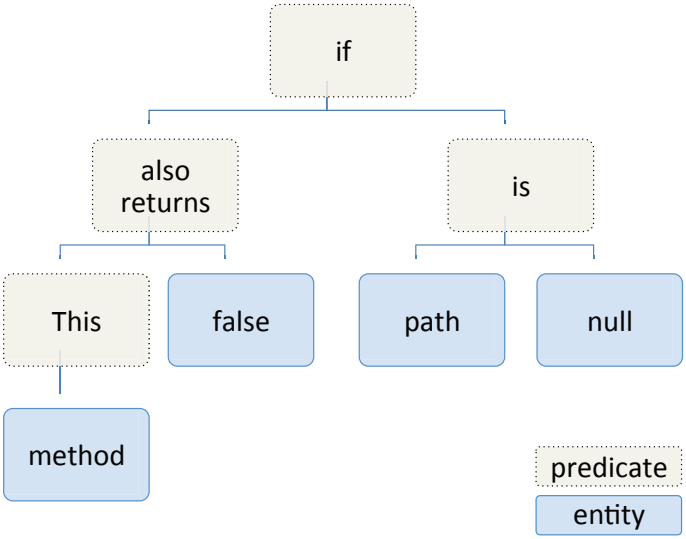

Figure 2.3: Intermediate representation of comment "This method also returns false if path is null"

'*path*' is the governing entity, '*null*' is the dependent entity, and these entities are related by predicate '*is*'.

CHAPTER 3

Inferring Method Specifications from Natural Language API Descriptions

This chapter introduces an approach to infer method arguments and return values specifications. [1] These specifications describe how to use a particular method within a class in terms of the expectations of the method arguments and expected return values. The remainder of this chapter is organized as follows. Section 3.1 introduces the approach. Section 3.2 presents an two real world examples motivating the approach. Section 3.3 presents the approach. Section 3.4 presents evaluation of the approach. Section 3.5 presents a brief discussion and future work.

3.1 Introduction

Software reuse is commonly practised since the advent of software development [FK05]. Software reuse facilitates development of larger, more complex, and timely-delivered software systems [GC92]. 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

[1] This work was done in collaboration with Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Parts of this Chapter appreared in proceedings of of the 34th International Conference on Software Engineering (ICSE 2012) [PXZ+12]

inconsistent with the expectations of libraries. As a result, not only such code is of inferior quality and contains faults, the added cost of debugging and correcting such faulty code could also defeat the purpose of reusing software.

Code contracts [Mey92, Cha06] 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# [BLS04], JML[2], and Code Contracts for .NET[3]. Additionally, code contracts can be used for formal proofs and automated code correction [WPF+10].

Despite being highly desirable, code contracts do not exist in a formalized form in most existing software systems in practice [PCM09]. 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, Oracle's J2EE API documentation[4] is an example of online documentation for Java programming language.

Even with such API documents, client-code developers often overlook some API descriptions and use methods in API libraries incorrectly [NW06]. 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 into formal specifications.

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

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

Ambiguity. Existing linguistic analysis techniques focus on well-written documents, such as news articles [SNL99]. In contrast, method descriptions are often not well-written. For instance, consider the sentence from the `File` class in the C# .NET Framework: *"true if path is an absolute path; otherwise false"*. This sentence does not have the main subject and the verb.

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

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

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

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

In summary, our approach infers specifications from existing API documents, thus facilitating verification of client code of an API library against the natural language descriptions of the library. As our approach analyzes API documents in natural language, it can be reused independent of the programming language of the library. Additionally, our approach complements existing approaches [WFKM11, CTS08, NE02] that infer code contracts from source code or binaries. To the best of our knowledge, ours is the first approach that analyzes API documents to infer specifications targeted towards generating code contracts.

This chapter makes the following major contributions:

- An approach that effectively identifies natural language sentences (in the API documents for a library) that describe code contracts, hereby referred to as contract sentences, and a

technique to infer specifications from the identified contract sentences.

- A prototype implementation of our approach based on extending the Stanford Parser [KM03a, SNL99], which is a natural language parser to derive the grammatical structure of sentences. An open source implementation of our prototype can be found at our website[5].

- An evaluation of our approach on 2717 sentences (for 333 methods) in five different classes from the .NET Framework and Facebook API for C#. Our evaluation results show that our approach effectively identifies contract sentences with an average of 92% precision and 93% recall. Additionally, our approach infers specifications from 1691 contract sentences with an average accuracy of 83%.

3.2 Motivating Examples

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

NullPointerException. As shown in Figure 3.1, a developer asked a question (on `06-30-2010`) in one of the Java forums[6] regarding `java.lang.NullPointerException`. Figure 3.1 shows the code snippet. After going through the suggestions from the contributing forum members, the author of this question was finally able to resolve the problem to a path issue occurring in the underlined line, a month (`07-30-2010`) after he initially posted the question.

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

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

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

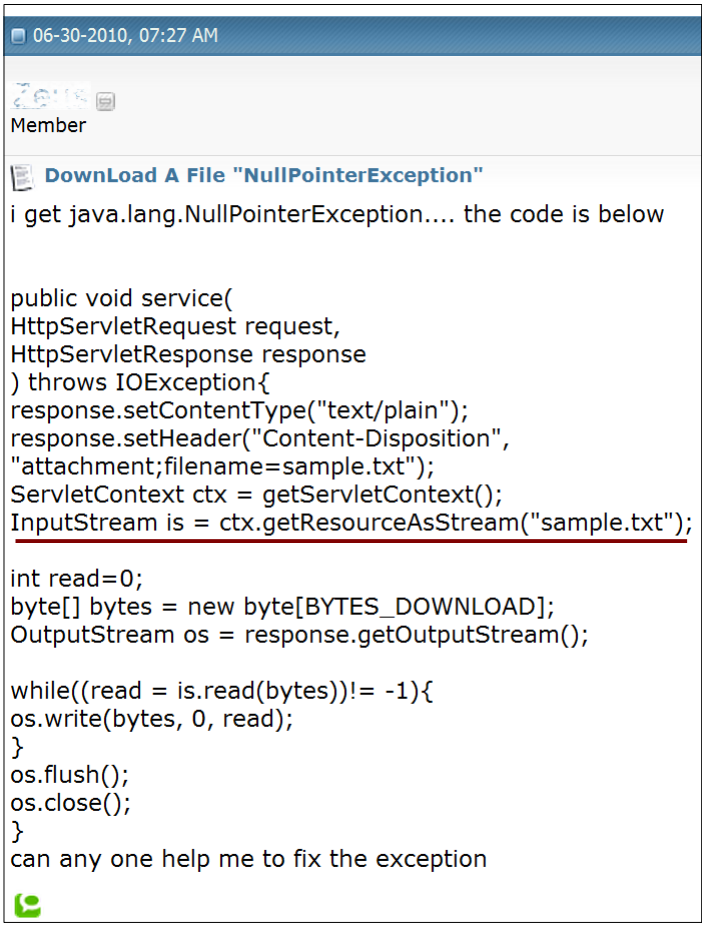

Figure 3.1: Description of a question posed in a Java forum

operation on the return value of the `getResourceAsStream` method, thus asking the developer to put the necessary checks in place.

Path format not supported. Another developer posted a 44-line code snippet in a forum[7]. The developer reported that he/she was noticing exceptions while moving a file. After a brief exchange of messages, the issue was traced to the `moveTo` method in the `FileInfo` class in C# .NET. Further exchanges revealed that the developer was attempting to move and rename the file to "11-19-2009_5:48:27.txt". The method description for the `moveTo` method in `FileInfo` states that the method throws an `ArgumentException` if the new file name consists of invalid characters defined in `Path.getInvalidCharacters()` (including':' character). After our approach infers a code contract from the description, the problem can be identified and reported by static checkers that are usually built into IDEs.

3.3 Approach

We next present our approach for inferring specifications from method descriptions. Figure 3.2 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 desired descriptive texts 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. We next describe these components in detail.

3.3.1 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;

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

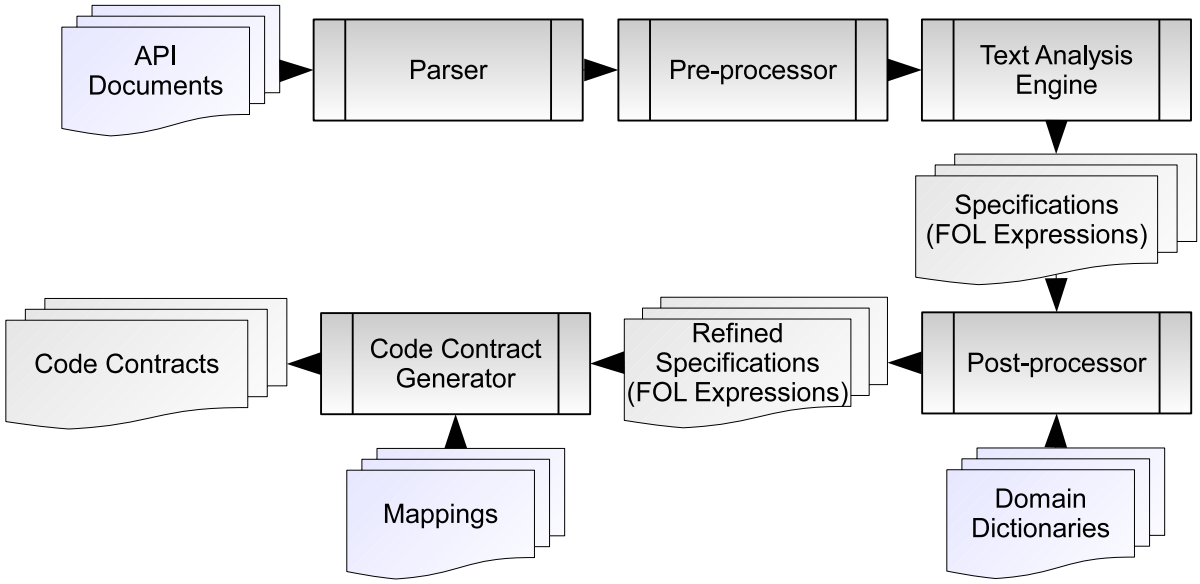

Figure 3.2: Overview of our approach

(4) *exception description*: the descriptions of exceptions explicitly thrown by the method; (5) *remark description*: additional descriptions about the functionality of the method.

3.3.2 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 3.3, 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 placeholder and matching a template for code contracts using the data type of the variable. In particular, the pre-processor uses method signatures and their associated tags for meta-data augmentation. From the method signatures, our approach extracts the name of the

```
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 3.3: The method description of the `DefineObjectProperty` method in Facebook API

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 [HFB+08] 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.

3.3.3 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 [SPKB09, SSP10] shows that FOL is an adequate representation for natural language analysis.

We first use a POS tagger to annotate POS tags in a sentence. We then use an NLP technique, called shallow parsing [Bog00]. 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 [Bog00, ST97, SPKB09, Gre99] 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 3.1 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 3.1, 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 3.1 to analyze the sentence in Line 3 of Figure 3.3. Figure 3.4 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 [KM03a], 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.

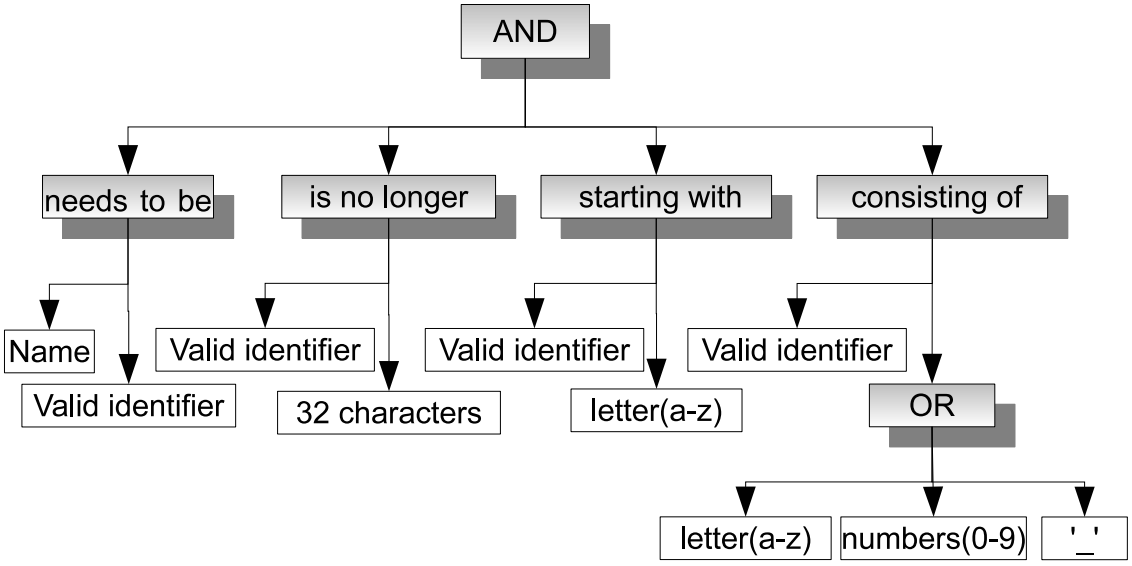

Figure 3.4: Specifications in format of FOL expressions extracted by our NLP Parser for `DefineObjectProperty` method in Facebook API

Table 3.1: 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.

3.3.4 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 3.4. 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 3.4 can also be represented as "begins with". Our post-processor identifies and classifies all semantically equivalent predicates into a single category, and thus reduces the effort to individually write mappings for every predicate in inferred FOL expressions even when they represent same the semantic function. We have identified the following seven major semantic categories for predicates: (1) Greater, (2) Lesser, (3) Begin, (4) End, (5) Consist, (6) Equal, and (7) Action with respect to expressions dealing with code contracts. The negative semantic categories are represented using a negation operator preceding the identified semantic class.

In the preceding semantic analysis, our post-processor uses an NLP technique called lemmatization [SNL99]. 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 [ea98] 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 3.3. 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 3.4. 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 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**.

Algorithm 1 Expression Augmentation generator

Require: Expr e, Meta-data d
Ensure: Expr e'

```
1:  Expr e' = e
2:  if (d.description == return) then
3:      if (e'.root == " → ")&&(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 == " → ")&&(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! == " → ") then
20:         Term t = d.exception_name
21:         Predicate p = new Predicate ("throw")
22:         p.term = t
23:         Expr e'' = new Expr(" → ")
24:         e''.left = e'
25:         e''.right = p
26:         e' = e''
27:     end if
28: end if
29: return e'
```

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.

3.3.5 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.

For example, consider the FOL expression in Figure 3.5. 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

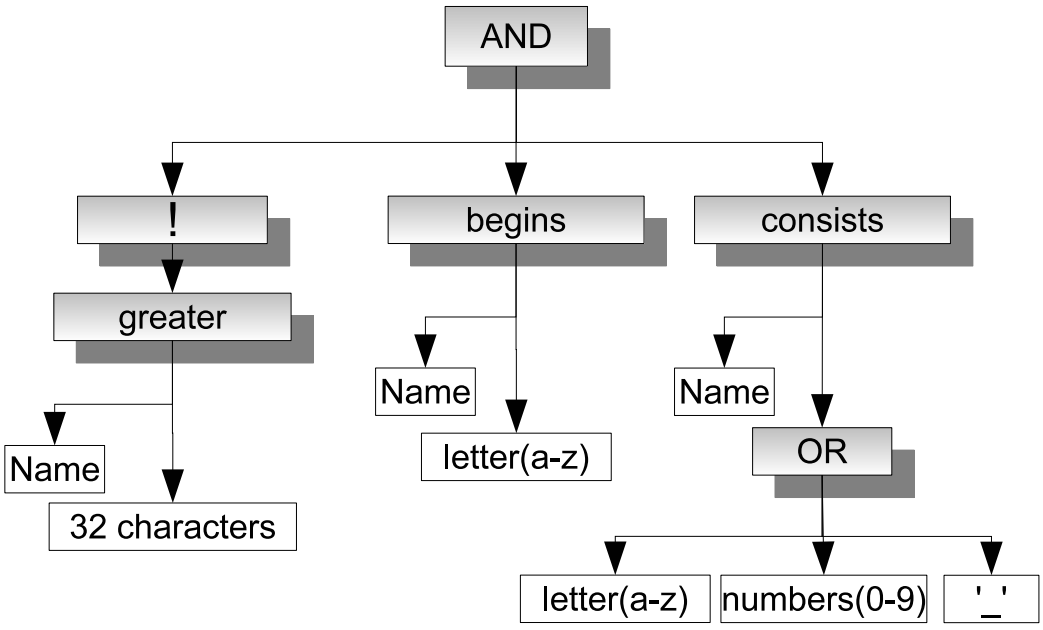

Figure 3.5: FOL expression after synonym analysis and compaction for the `DefineObjectProperty` method in Facebook API

(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]+"))`.

3.4 Evaluation

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

- **RQ1**: What are the precision and recall of our approach in identifying contract sentences (i.e., sentences that describe code contracts)?

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

Figure 3.6: The inferred specifications for the `prop_name` variable of the `DefineObjectProperty` method in Facebook API

- **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?

3.4.1 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[8]. Due to the sheer popularity of Facebook and a huge number of developers developing third-party applications, we chose the Facebook API [9]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 3.2 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.

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

3.4.2 Evaluation Results

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 researchers independently labeled sentences as contract sentences by discussing iteratively until they reached a consensus. We then applied our approach on the API documents and manually measured the number of `true positives` (TP), `false positives` (FP), and `false negatives` (FN) produced by our approach as follows:

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

In statistical classification [Ols08], 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 3.2 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.

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 3.2 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%.

Table 3.2: 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

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%.

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. [RB10] 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 3.2 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"* [RB10]. 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 3.3, 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 3.6. Furthermore, no direct text (in the method description) could be found 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 3.3 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 [CTS08, NE02].

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 3.2 from over 2717 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 3.2 from over 1691 contract sentences. Furthermore, our evaluation results show that our approach can infer detailed specifications than human written contracts.

3.4.3 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# [RB10], 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.

3.5 Discussion and Future work

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

Code Searching. Code searching [TX07, Rei09] for reuse is a classic problem [FK05] in software engineering. Among previous approaches, a recent approach by Riess [Rei09] provides promising results by using semantics such as code contracts as input-output relationships for code searching. Our approach can be used for generating specifications from API documents in a code repository and thus assisting such approaches in producing better results.

Program Synthesis. Automated program synthesis holds potential for easing the task of a developer by taking care of program generation and allowing the developer to concentrate on design tasks. Recent work by Srivastava et al. [SGF10a] addresses the problem by leveraging specifications in the form of pre/post-conditions and invariants to achieve synthesis. Our approach can work in conjunction with such approaches to extract specifications from natural language text to achieve better synthesis.

3.5.1 Limitations:

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

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

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

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

Validation of Method Descriptions. API documents can sometimes be misleading [TMTL12a, RGL10], thus causing developers to write faulty client code. In future work, we plan to extend our approach to find documentation-implementation inconsistencies.

Elimination of Predefined Lists. The current implementation of our approach uses pre-defined lists for domain dictionaries. There are approaches [ZCY08] that facilitate building domain dictionaries from source code. We plan to extend our implementation to use these approaches. Furthermore, we rely on pre-defined templates for code contract generation. While such a strategy serves our purpose of prototyping, advanced techniques such as keyword programming [LM07] have shown promising results in building programming statements using keywords. We plan to explore such techniques and evaluate the overall effectiveness of our approach after augmenting it with such techniques.

3.6 Chapter Summary

Specifications described in natural language in API documents are not amenable to formal verification by existing verification tools. In this chapter, we have presented a novel approach for inferring formal specifications from API documents targeted towards code contract generation. Our evaluation results show that our approach has an average of 92% precision and 93% recall in identifying sentences describing code contracts from over 2717 sentences. Furthermore, our results also show that our approach has an average of 83.4% accuracy in inferring specifications from sentences describing code contracts out of over 1691 sentences.

CHAPTER 4

ICON: Inferring Temporal Constraints from Natural Language API Descriptions

Temporal constraints of an Application Programming Interface (API) are the allowed sequences of method invocations in the API. These constraints govern the secure and robust operation of client software using the API. This chapter introduces ICON: an approach to infer temporal constraints. [1] The remainder of this chapter is organized as follows. Section 4.1 introduces the approach. Section 4.2 presents a real-world example that motivates our approach. Section 4.3 presents our approach. Section 4.4 presents the evaluation of our approach. Section 4.5 presents brief discussion on the limitations and future work.

4.1 Introduction

Application Programming Interfaces (APIs) facilitate software reuse by providing a standardized mechanism to access API components. However, an API has some constraints (governing the proper use of the API) that must be followed. One such type of constraints are temporal constraints [2], which are the allowed sequences of invocations of methods from the API. Non-compliance to such constraints often results in faulty applications that are unreliable.

[1] This work was done in collaboration with Kunal Taneja, Tao Xie, Laurie Williams, and Teresa Tung

Formal analysis tools, such as model checkers and runtime verifiers, can assist in detecting violations of the temporal constraints in API clients as defects [LJMR12]. These tools typically accept a formal representation of the temporal constraints for detecting violations. However, temporal constraints are typically described in natural language text of API documents. Such documents are provided to client-code developers through an online access, or are shipped with the API code. For a method under consideration, an API document may describe both the constraints on the method parameters as well as the temporal constraints in terms of other methods that must be invoked pre/post invoking that method.

Although natural language API descriptions can be manually converted to formal constraints, manually identifying and writing formal constraints based on natural language text in API documents can be prohibitively time-consuming and error-prone [WWL+13, RB10]. For instance, the PDF version of the documentation for the `Amazon S3 REST` API[2] spans 278 pages describing 51 methods.

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

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

For instance, consider the sentences from the `PayPal Payment REST` API: 1) "*Use this call to complete a payment.*" from the `execute payment` method; 2) "*Use this call to refund a completed payment.*" from the `refund sale` method. Sentence 1 is a descriptive statement about the `execute payment` method. In contrast, Sentence 2 indicates the temporal constraint that a payment must be completed before the refund call is initiated. Since these sentences are not significantly different in terms of words, a simple keyword-based search will fail to distinguish between the two. Another problem with keyword-based search is that the method names (and synonyms) are part of the keywords themselves, thus resulting in a large number of keywords to be searched. The large number of keywords further negatively affects accuracy.

Even after a sentence has been identified as a constraint sentence, an approach still needs to infer the references to the method in the sentences, which often may not be explicit. Consider

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

Sentence 2 in the preceding example. The phrase *"completed payment"* refers to the `execute payment` method in the API.

To address these issues, we propose ICON: an approach based on Machine Learning (ML) and Natural Language Processing (NLP) for identifying and inferring formal temporal constraints. We propose to first employ ML for identifying temporal constraint sentences and then use NLP techniques to infer formal temporal constraints from the identified sentences.

An ML-based approach for identifying the temporal constraint sentences addresses the limitations of keyword-based search by automatically learning patterns of temporal constraint sentences. Furthermore, we propose to use a combination of words, syntax (parts of speech) of a sentence, and semantics (relationship between words) of a sentence as the features for learning patterns. This combination allows ICON to make a finer-grained distinction between the example sentences described earlier. Furthermore, to identify phrases as implicit method-invocation references, we propose to leverage domain dictionaries that are systematically created from API documents and generic English dictionaries.

In summary, the ICON approach leverages the natural language description of an API to infer temporal constraints of method invocations. As our approach analyzes API documents in natural language, it can be reused independent of the programming language of an API library. Additionally, our approach complements existing mining-based approaches [BW12, TX07, WDZ+13, ZXZ+09] that partially address the problem by mining for common usage patterns among client code reusing the API. Our results indicate that ICON is effective in identifying temporal constraint sentences (from over 4000 human-annotated API sentences) with the average precision, recall, and F-score of 79.0%, 60.0%, and 65.0%, respectively. Furthermore, our evaluation also demonstrates that ICON achieves an accuracy of 70% in inferring 77 formal temporal constraints from these sentences. This chapter makes the following main contributions:

- An ML- and NLP-based approach that effectively infers formal temporal constraints of method invocations.
- A prototype implementation of our approach based on extending the Stanford Parser [KM03b], which is a natural language parser to derive the grammatical structure of sentences.
- An evaluation of our approach on the `Amazon S3 REST` API, the `PayPal Payment REST` API, and the commonly used package `java.io` from the JDK API. All our experimental subjects, results, and artifacts are publicly available on our project website. [proa]

4.2 Motivating Example

We next present a real world example to motivate our approach. Through the example, we demonstrate that developers often ignore the temporal constraints of an API described in the documentation. We suspect the reason for this behavior is that the documentation is often verbose and the information is distributed across various pages. Often developers may not have time (and/or patience) to go through all the documentation and may overlook some temporal constraints of the API, resulting in defective client applications that invoke API methods in sequences prohibited by documentation.

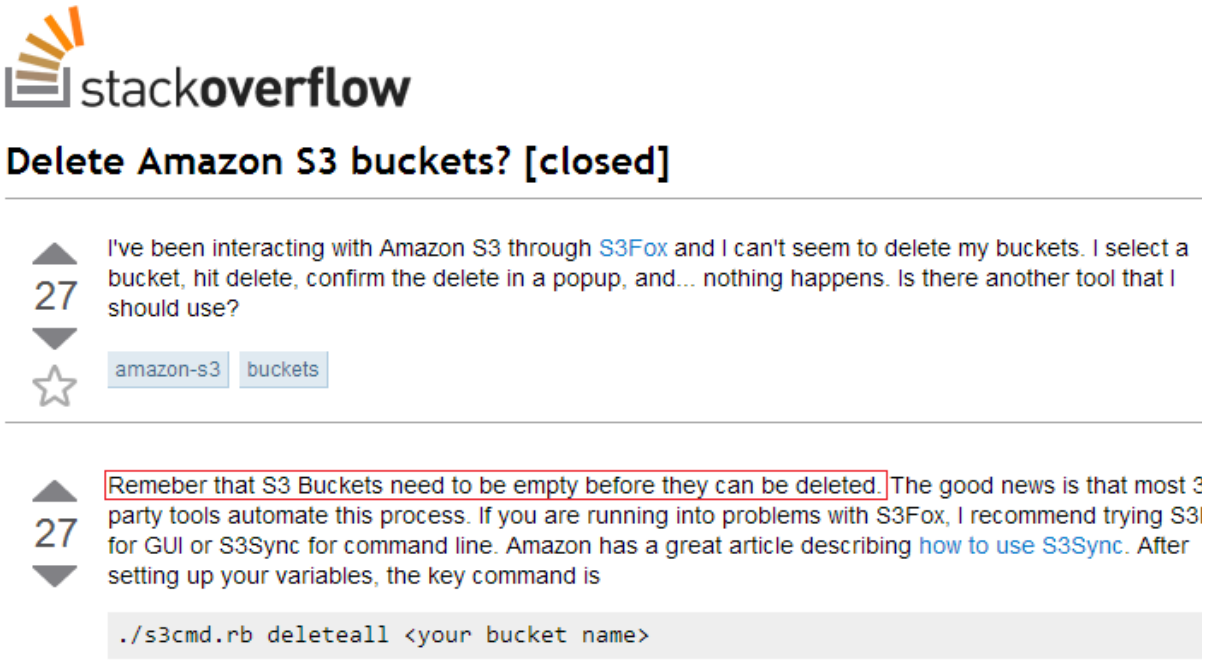

Figure 4.1: Query posted on StackOverflow forum about Amazon S3 REST API

Consider the question asked in *Stack Overflow* [3] as shown in Figure 4.1. Stack Overflow is an online question and answer forum for professional and enthusiast programmers. The query is about the delete functionality of a third-party software `S3Fox` to interact with `Amazon S3 REST` API. The inquisitor complains about an issue in the delete bucket functionality of the

[3] `http://stackoverflow.com/questions/27267/delete-amazon-s3-buckets`

`S3Fox`. The `S3Fox` developers overlooked the constraints in `Amazon S3 REST` API developer documents, causing the issue. The API document pertaining to the delete bucket functionality states that before deleting the bucket, the objects in the buckets must be deleted. *"All objects (including all objects versions and Delete Markers) in the bucket must be deleted before the bucket itself can be deleted"*. Although the issue was fixed, one of the forum responses recommended the inquisitor to switch to another product. Customer dissatisfaction caused by such issues with the delete bucket functionality can lead to a loss in revenue.

The presented issue can be easily detected using formal analysis tools. For instance, a specification rule (temporal constraint) can be added to a static checker to verify the presence of a call to delete object functionality before the call to delete bucket functionality.

4.3 ICON Overview

We next present our approach for inferring temporal constraints from the method descriptions in API Documents. Figure 4.2 gives an overview of the ICON approach. ICON consists of six major components: a preprocessor, a candidate identifier, a text-analysis engine, a semantic graph generator, constraint extractor, and a type analyzer. Additionally, our approach uses an optional model trainer component and external NLP parser component.

First, the preprocessor accepts API documents and preprocesses the sentences in the method description. Next, an NLP parser annotates the syntax and semantics of preprocessed sentences. The annotated sentences are accepted by candidate identifier component to classify temporal constraint sentences, using the model trained by the model trainer component. The text-analysis engine further transforms the identified constraint sentences into the first-order-logic (FOL) representation. Finally, the constraint extractor leverages the semantic graphs to infer temporal constraints from the FOL representation of a sentence. The type analyzer component infers temporal constraints encoded in the type system of a language by analyzing the API methods parameter and return types. We next describe each component in detail.

4.3.1 Preprocessor

The preprocessor accepts the API documents and extracts method descriptions. In particular, the preprocessor extracts the following fields from the method descriptions: 1) summary of the API method; 2) summary and type information of parameters of the API method; 3) summary

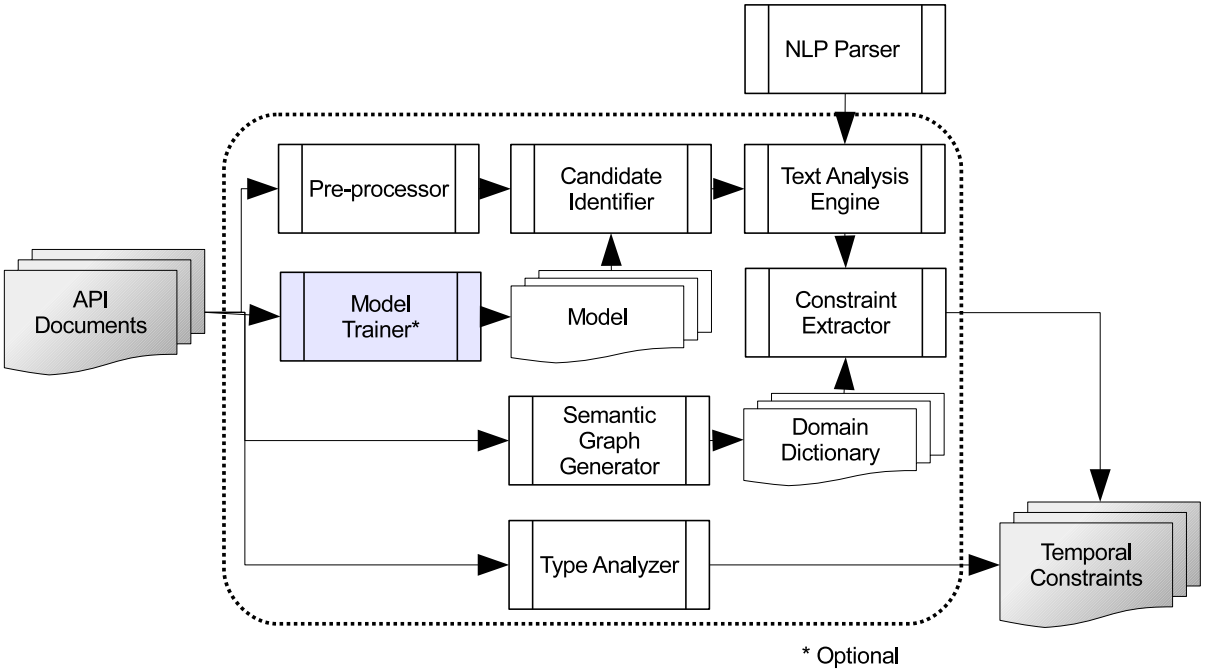

Figure 4.2: Overview of the ICON approach

and type information of return values of the method; and 4) summary and type information of exceptions thrown (or errors returned) by the methods.

This step is required to extract the desired descriptive text from the API documents. Different API documents may have different styles of presenting information to developers. This difference in style may also include the difference in the level of the details presented to developers. ICON relies only on basic fields that are generally available for API methods across different presentation styles.

After extracting desired information, the natural language text is further preprocessed to be analyzed by subsequent components. The preprocessing steps are required to increase the accuracy of core NLP techniques (described in Chapter 2) that are used in the subsequent phases of the ICON approach. The accuracy of core NLP techniques is often inversely proportional to the number of lexical tokens in a sentence. Thus, reduction in the number of lexical tokens greatly improves the accuracy of core NLP techniques. The preprocessor employs following heuristics to achieve the desired reduction of lexical tokens:

- *Named Entity Handling*: Sometimes a sequence of words correspond to the name of entities that have a specific meaning collectively. For instance, consider the phrases *"Amazon*

S3", "Amazon simple storage service", which are the names of the service. Further resolution of such phrases using grammatical syntax is redundant for inference needs. This heuristic annotates the phrase representing the name of an entity as a single lexical token, to improve accuracy of core NLP techniques.

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

4.3.2 NLP Parser

The NLP parser accepts the pre-processed documents and annotates every sentence in each document using core NLP techniques [dMMM06, dMM08, PXZ+12, PXY+13, TSSC12] described in Chapter 2. In particular, each sentence is annotated with: *1) POS tags, 2) named-entity annotations, and 3) Stanford-typed dependencies.*

Next we use an example to illustrate the annotations added by the NLP Parser. Consider the sentence from the example section ***'All objects (including all object versions and Delete Markers) in the bucket must be deleted before the bucket itself can be deleted."***. Figure 4.3 shows the sentence annotated by NLP parser. Each word (occurs first) is followed by the Part-Of-Speech (POS) tag of the word (in green), which is further followed by the name of Stanford dependency connecting the actual word of the sentence to its predecessor. From an implementation perspective, this component can be implemented using any existing NLP libraries or approaches such as Stanford Parser [SNL99].

4.3.3 Candidate Identifier

This component accepts the annotated sentence from the previous component, then using a trained ML model, classifies whether each sentence as a temporal constraint sentence or not. We next describe the model construction.

```
-> deleted-VBN (root)
  -> objects-NNS (nsubjpass)
    -> All-DT (det)
    -> including-VBG (dep)
      -> object versions-NNS (pobj)
        -> all-DT (det)
        -> Delete Markers-NNS (conj_and)
      -> Delete Markers-NNS (pobj)
    -> bucket-NN (prep_in)
      -> the-DT (det)
  -> must-MD (aux)
  -> be-VB (auxpass)
  -> bucket-NN (prep_before)
    -> the-DT (det)
    -> deleted-VBN (rcmod)
      -> itself-PRP (nsubjpass)
      -> can-MD (aux)
      -> be-VB (auxpass)
```

Figure 4.3: Sentence annotated with Stanford dependencies

The goal of model construction is to use a small set of manually classified temporal constraint sentences of a representative API classify the unlabeled sentences as being temporal constraints or not. From an implementation perspective, we can use any of the standard off-the-shelf classifiers to train the ML model, since ICON seeks to achieve only a binary classification of the API sentences. The ML model can be trained offline or else by the users of the approach if better accuracy is desired for domain-specific data.

Feature selection is an important factor for the accuracy any classification method. In the simplest form, each word occurring in a sentence is considered as a feature. However, such an approach may lead to overly specific ML models, and therefore the ICON approach extracts generic features from sentences. We next describe the features we chose for training our ML model along with the rationale for selecting such features.

1. **Length of a Sentence**: The feature is the total number of words in the sentence. The rationale is that shorter sentences (containing fewer words) are unlikely candidates for temporal constraint sentences.

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

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

4. **Stopword Reduction**: In linguistics, stopwords are the frequently occurring words that can be ignored and are often considered noise, such as "the" , "of", "to" etc... The rationale for filtering the stopwords is to reduce the size of feature set that otherwise considers such words (despite being noise) as an independent features. Stopword list is further augmented by adding to them the words that occur exactly once in the training corpus to avoid over-fitting of the model to the training corpus.

5. **Stanford Dependencies**: We add to the feature set by identifying the presence of specific Stanford-typed dependencies pertaining to the temporal aspects of the sentence semantics. In particular, we identify the presence of "*advcl*", "*aux*", "*auxpass*", "*vmod*", and "*tmod*". For instance, the annotated sentence in Figure 4.3 contains both "*aux*" and "*auxpass*" dependencies. These are both added to the feature set.

6. **POS Tags**: We filter words whose part of speech tags are "*Noun*", "*Determiner*", "*Adjective*", "*Cardinal Number*", "*Foreign Word*", "*Brackets*", "*Coordinating Conjunction*", and "*Personal Pronouns*". The rationale for filtering based on POS tags is to remove the words that are unlikely to be specific to temporal constraints. For instance, presence or absence of determiners is unlikely to affect the outcome of classifier. We further, annotate the POS of word to further distinguish between the words used in different context.

7. **Sentence Structure**: The feature is the ordered sequence of chunking tags [KM03b, KM03c] in a sentence. Chunking seeks to divide a sentence into a constituent set of words (or phrases) that logically belong together (such as a Noun Phrase and Verb Phrase). Thus chunking captures the structure of a sentence. Rationale for selecting ordered sequence of chunking tags is to incorporate structure of a sentence as a feature.

We use the described feature set to train a classifier to perform binary classification of sentence as either temporal constraint describing or not. The rationale of using ML based approach as opposed to a rule based approach is because: 1) rule-writing requires domain expertise; 2) rules-writing tends to quickly become ad hoc thus requires greater effort to generate generic rules to avoid over-fitting; and 3) ML classifiers have shown to scale well with large volumes of data.

4.3.4 Text Analysis Engine

The text analysis engine component accepts the sentences identified as constraint sentences and creates an intermediate representation of each sentence. This intermediate representation is leveraged by subsequent component to infer formal constraints. We define our representation as a tree structure that is mimics a FOL expression. Research literature provides evidence of the adequacy of using FOL for NLP related analysis tasks [SPKB09, SSP10, PXZ+12, PXY+13].

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.

Chapter 5 introduces the intermediate representation generation technique implemented as a function of Stanford-typed dependencies [dMMM06, dMM08, KM03c], to leverage the semantic information encoded in Stanford-typed dependencies. However, we observed that such implementation is overwhelmed by complex sentences. We improve the accuracy of intermediate-representation generation by proposing a hybrid approach, i.e. taking into consideration both the POS tags as well as Stanford-typed dependencies. The POS tags which annotate the syntactical structure of a sentence are used to further simplify the constituent elements in a sentence. We then use the Stanford-typed dependencies that annotate the grammatical relationships between words to construct our FOL like representation. Thus, the intermediate representation generator used in this work is a two phase process as opposed to single phase in previous work [PXZ+12, PXY+13]. We next describe these two phases:

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

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

The Stanford parser faces difficulty to annotate accurately the Stanford-typed dependencies of the sentence because of presence of various clauses acting on various subject-object pairs. As shown in Figure 4.3 the word "including" is annotated with Stanford-typed dependencies "dep" that is a catch all dependency. A catch all dependency is selected by a parser when no other appropriate dependency can be selected. The ICON approach thus automatically break down the sentence into two smaller tractable sentences, which are accurately annotated:

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

Table 4.1 lists the semantic templates used in this phase. Column "Template" describes conditions where the template is applicable and Column "Summary" describes the action taken by ICON when the template is applicable. With respect to the previous example the template 3 *(A noun phrase followed by another noun/pronoun/verb phrase in brackets)* is applicable. Thus our shallow parser breaks the sentence into two individual sentences.

• **Stanford-typed Dependencies**: After complex sentences (whenever applicable) have been broken to simple sentences, this phase generates an intermediate (FOL) representation of the sentences. This phase accepts the syntax-annotated sentences (grammatical and semantic) and builds a First-Order-Logic (FOL) like representation of the sentence. Earlier researches have shown the adequacy using FOL for NLP related analysis tasks [SPKB09, SSP10, PXZ+12]. The component is implemented as a sequence of cascading finite state machines based on the function of annotated Stanford-typed dependencies [dMMM06, dMM08, KM03b, KM03c].Figure 4.4 is the FOL representation of the sentence *"All objects in the bucket must be deleted before the bucket itself can be deleted"*. All the leaf nodes (entities) are represented as the bold words. For readability each node is appended with a number representing the in-order traversal index of the tree.

Table 4.1: Semantic Templates

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

```
01:before[6]
02:|->must be deleted[5]
03:   |->All[1]
04:   |   |->in[3]
05:   |      |->objects[2]
06:   |      |->bucket[4]
07:   |->can be deleted[8]
08:      |->bucket[7]
09:      |->itself[9]
```

Figure 4.4: FOL-like representation of the sentence *"All objects in the bucket must be deleted before the bucket itself can be deleted."*

4.3.5 Constraint Extractor

Constraint Extractor is responsible for inferring temporal constraint from the classified constraint sentences. Temporal constraints are expressed as temporal formulae involving: a) *Predicates* ξ representing method calls and b) *Temporal operators*: backward ($\leftarrow$) & forward ($\rightarrow$) and their negations $\overleftarrow{-}$ & $\overrightarrow{-}$. We define the following four constraints:

1. *Forward Operator ($\xi_1 \rightarrow \xi_2$)*: method call ξ_1 must be succeeded by method call ξ_2.
2. *Backward Operator ($\xi_1 \leftarrow \xi_2$)*: method call ξ_1 must be preceded by method call ξ_2.
3. *Negative Forward Operator ($\xi_1 \overrightarrow{-} \xi_2$)*: method call ξ_1 cannot be succeeded by method call ξ_2
4. *Negative Backward Operator ($\xi_1 \overleftarrow{-} \xi_2$)*: method call ξ_1 cannot be preceded by the method call of ξ_2

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

ξ_1: ICON first identifies ξ_1 as the method whose description the constraint sentence is part of. For instance the sentence in Figure 4.3 is part of Delete Bucket method description in `Amazon S3 REST` API, ξ_1 is instantiated as Delete Bucket method.

ξ_2: ICON next identifies ξ_2. Since, references to ξ_2 may not always be implicit we leverage the semantic graphs. A semantic graph is a representation of objects and the methods applicable on those objects. Figure 4.5 shows a graph for `Object` resource in `Amazon S3 REST` API.

Algorithm 2 Action_Extractor

Require: K_Graph g, FOL_rep rep
Ensure: String $action$

```
1: String action = φ
2: List r_name_list = g.resource_Names
3: FOL_rep r' = rep.findLeafContaining(r_nam_list)
4: List actionList = g.actionList
5: while (r'.hasParent) do
6:    if actionList.contains(r'.parent.predicate) then
7:       action = actionList.matching(r'.parent.predicate)
8:       break
9:    else
10:      if actionList.contains(r'.leftSibling.predicate) then
11:         action = actionList.matching(r'.leftSibling.predicate)
12:         break
13:      end if
14:   end if
15:   r' = r'.parent
16: end while
17: return action
```

The phrases in rounded rectangle are the actions applicable on `Object` resource. Section 4.3.6 further describes how these graphs are generated. ICON uses the Algorithm 2 to identify ξ_2.

ICON systematically explores the intermediate representation of the candidate sentence to identify ξ_2. First, this component attempts to locate the occurrence of object name or its synonym in the leaf nodes (entity) of the intermediate representation of the sentence. Once a leaf node is found, this component systematically traverses the tree from the leaf node to the root, matching all parent predicates as well as immediate child predicates. This component matches each of the traversed predicate with the actions associated with the object defined in semantic graph. ICON further employs WordNet and Lemmatisation to deal with synonyms to find appropriate matches. If a match is found, then the matching action is returned as ξ_2. ICON does not consider self references, that is if $\xi_2 = \xi_1$, the identified method reference is discarded. In case of multiple matching actions ICON considers only the first match. For instance the sentence in Figure 4.3 algorithm identifies Delete Object method as ξ_2

Temporal Operator: ICON next identifies the direction (forward or backward) of the relationship by examining the tense of ξ_2 reference in the sentence. Past tense is considered as

backward and other tenses are considered as forward. For instance, the sentence in Figure 4.3 since "deleted" is in past tense and therefore operator is backward and the constraint is Delete Bucket ← Delete Object. Negation is identified by presence of the negation verbs such as "*no*", "*not*", "*can't*" ... etc. Another rule for negation operator is if the sentence is in exception/error description.

4.3.6 Semantic-Graph Generator

A key way of identifying reference to a method in the API by ICON is the employment of a semantic graph of an API. We propose to initially infer such graphs from API documents. Ad hoc creation of semantic graph is prohibitively time consuming and may be error prone. We thus employ a systematic methodology (proposed previously in Whyper [PXY+13]) to infer semantic graphs from API documents.

We first consider the name of the class for the API document in question. We then find the synonyms terms used refer to the class in question. The synonym terms are listed as by splitting the camel-case notation in the class name. This list is further augmented with the name of the parent classes and implemented interfaces if any. We then systematically inspect the member methods to identify actions applicable to the objects represented by the class. From the name of a public method (describing a possible action on the object), we extract verb phrases. The verb phrases are used as the associated actions applicable on the object. In case of REST API we first identified the resources and then listed REST actions on those resources as applicable actions. Figure 4.5 shows the graph for `Object` resource in REST API. The phrases in rounded rectangle are the REST actions applicable on `Object` resource in `Amazon S3 REST` API.

4.3.7 Type Analysis

Some temporal constraints are enforced by the type system in typed Languages. For instance, a method (m) accepting input parameter (i) of type (t) mandates that (at least one) method (m') be invoked whose return value is of type (t). To extend the temporal constraints inferred by the analyzing the natural language text, this component infers additional constraints that are encoded in the type system. Algorithm 3 lists the steps followed to infer type based temporal constraints.

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

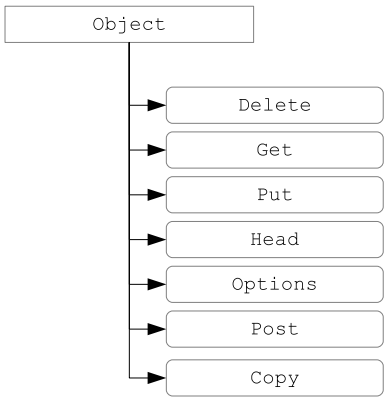

Figure 4.5: Semantic Graph for the `Object` method in Amazon S3 REST API

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

4.4 Evaluation

We next present the evaluation we conducted to assess the effectiveness of ICON. In our evaluation, we address three main research questions:

- **RQ1**: What are the precision and recall of ICON in identifying natural language sentences describing temporal constraints? Answer to this question quantifies the effectiveness of ICON in identifying constraint sentences.
- **RQ2**: What is the accuracy of ICON in inferring temporal constraints from constraint sentences in the API documents? Answer to this question quantifies the effectiveness of

Algorithm 3 Type_Sequence_Builder

Require: List $methodList$
Ensure: Graph seq_Graph

```
1: Graph seq_Graph = φ
2: Map idx = createIdx(methodList)
3: for all Method mtd in methodList do
4:    seq_Graph.addVertex(mtd)
5: end for
6: for all Method mtd in methodList do
7:    if mtd.isPublic() then
8:       if !mtd.isStatic() then
9:          List preList = idx.query(mtd.declaringType)
10:         for all Method mtd' in preList do
11:            seq_Graph.addEdge(mtd', mtd)
12:         end for
13:      end if
14:      for all Parameter param in mtd.getParameters() do
15:         if !isBasicType(param.Type) then
16:            List preList = idx.query(paramType)
17:            for all Method mtd' in preList do
18:               seq_Graph.addEdge(mtd', mtd)
19:            end for
20:         end if
21:      end for
22:   end if
23: end for
24: return seq_Graph
```

ICON in inferring temporal constraints from constraint sentences.

- **RQ3**: What is the degree of the overlap between the temporal constraints inferred from natural language text in comparison to the typed-enforced temporal constraints?

4.4.1 Subjects

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

- `Amazon S3 REST` API provides a REST based web services interface to store and retrieve data on the web. Furthermore, `Amazon S3` also empowers a developer with rich

set of API methods to access a highly scalable, reliable, secure, fast, and inexpensive infrastructure. `Amazon S3` is reported to store more than 2 trillion objects as of April 2013 and gets over 1.1 million requests per second at peak time [ama].

- `PayPal Payment REST` API provides a REST based web service interface to facilitate online payments and money transfer. `PayPal` reports to have handled $56.6 billion(USD) worth of transactions (total value of transactions) in just the third quarter of 2014.

- `java.io` : is one of the widely used packages in `Java` programming language. The package provides APIs for system input and output through data streams, serialization and the file system, which are one of the fundamental functionalities provided by any programming language.

We chose `Amazon S3`, `PayPal payment`, and `java.io` APIs as our subjects because they are widely used and contain relevant documentation.

4.4.2 Evaluation Setup

We first manually annotated the sentences in the API documents of the subject APIs. Two researchers [4] manually labeled each sentence (2417 total) in the Java API documentation as being a temporal constraint sentence or not. We used cohen kappa [Car96] score to statistically measure the inter-rater agreement. The `cohen kappa` score of the two researchers was .66 (on a scale of 0 to 1), which denotes a statically significant agreement [Car96]. After the researchers classified all the sentences, they discussed with each other to reach a consensus on the sentences they classified differently. We use these classified sentences as the golden set for evaluating effectiveness of ICON. Table 4.3 lists the subject statistics. Based on the discussions with regards to annotation of `java.io` API, I annotated the rest of the subject APIs.

To answer RQ1, we first measure the number of true positives (TP), false positives (FP), true negative (TN), and false negatives (FN) in identifying the constraint sentences by ICON. We define constraint sentence as a sentence describing a temporal constraints. We define the TP, FP, TN, and FN of ICON as follows:

1. TP: A sentence correctly identified by ICON as constraint sentence.

[4] Rahul Pandita and Kunal Taneja

2. FP: A sentence incorrectly identified by ICON as constraint sentence.

3. TN: A sentence correctly identified by ICON as not a constraint sentence.

4. FN: A sentence incorrectly identified by ICON as not a constraint sentence.

In statistical classification [Ols08], `precision` is defined as a ratio of number of true positives to the total number of items reported to be true, `recall` is defined as a ratio of number of true positives to the total number of items that are true. `F-Score` is defined as the weighted harmonic mean of `precision` and `recall`. Based on the calculation of TP, FP, TN, and FN of ICON defined previously we computed the `precision`, `recall`, and `F-Score` of ICON as follows:

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

We then use these measures to identify relative effectiveness of ICON by executing multiple classifiers on the annotated sentences. We tested the classifiers using a stratified n-fold cross-validation approach, using 10 as the value n (10-fold) as recommended by Han et al. [HK06]. The cross validation ensures that every sentence is used for training and testing, thus producing low bias and variance. In particular, we execute the classifiers in two different configurations. First we executed the classifiers using the words in the sentences as features and measure `precision`, `recall`, and `F-Score` as P_{wrd}, R_{wrd}, and F_{wrd}. We next executed the classifiers on the features proposed by the ICON approach and measure `precision`, `recall`, and `F-Score` as P_{ftr}, R_{ftr}, and F_{ftr}. We next calculate relative gain in `precision`, `recall`, and `F-Score` as P_{Δ} (P_{ftr}- P_{wrd}), R_{Δ} (R_{ftr} - R_{wrd}), and F_{Δ} (F_{ftr} - F_{wrd}). Higher values of P_{Δ}, R_{Δ}, and F_{Δ} are indicative of effectiveness of the constraint statements inferred using the features proposed by the ICON approach.

To answer RQ2, we manually verified the temporal constraints inferred from constraint sentences by ICON. However, we excluded the type-enforced temporal constraints described in Section 4.3.7. We excluded the type-enforced constraints because they are correct by construction and are by default enforced by modern IDE's such as the Eclipse. We then measure $accuracy$ of ICON as the ratio of the total number of temporal constraints that are correctly inferred by ICON to the total number of constraint sentences. Two authors independently verified the correctness of the temporal constraints inferred by ICON. We define the `accuracy`

Table 4.2: Evaluation Results (Identification)

S. No.	Trainer	P_{wrd}	R_{wrd}	F_{wrd}	P_{ftr}	R_{ftr}	F_{ftr}	P_{Δ}	R_{Δ}	F_{Δ}
1	AdaBoost	0.52	**0.81**	0.61	0.7	**0.78**	0.74	0.18	-0.03	0.13
2	NaiveBayes	0.49	0.67	0.56	0.74	0.65	0.69	0.25	-0.02	0.13
3	Balanced Winnow	0.77	0.76	**0.76**	0.78	0.77	**0.77**	0.01	0.01	**0.01**
4	Decision Tree	**0.82**	0.58	0.66	**0.92**	0.41	0.56	0.1	-0.17	-0.1
5	Winnow	0.19	0.41	0.15	0.8	0.51	0.59	**0.61**	0.1	**0.44**
6	MaxEnt	0.69	0.48	0.55	0.76	0.46	0.56	0.07	-0.02	0.01
7	c45	0.78	0.4	0.52	0.82	0.59	0.67	0.04	0.19	0.15
	Average	0.61	0.59	0.54	0.79	0.6	0.65	0.18	0.01	0.11

All values are average over 10-fold cross validation; P: Precision; R: Recall; F: F-Score; $_{wrd}$: No features used for training; $_{ftr}$: features used for training; $_{\Delta}$: improvement factor ($_{ftr}$ - $_{wrd}$)

of ICON as the ratio of constraint sentences with correctly inferred temporal constraints to the total number of constraint sentences. Higher value of `accuracy` is indicative of effectiveness of ICON in inferring temporal constraints from constraint sentences.

To answer RQ3, we counted the overlap in the temporal constraints inferred by ICON from the natural language text in API documents to the type-enforced temporal constraints inferred using the method in Section 4.3.7.We next present our evaluation results.

4.4.3 Results

RQ1: Effectiveness in Identifying Constraint Sentences

In this section, we quantify the effectiveness of ICON in identifying constraint sentences by answering RQ1. Table 4.2 shows the effectiveness of ICON in inferring temporal constraints from the identified constraint sentences using various classifiers. Column "Trainer" lists the names of the classifiers used to train the model for classifications in ICON. Columns P_{wrd}, R_{wrd}, and F_{wrd} list the `precision`, `recall`, and `F-score` of classifiers trained without the features proposed by ICON. Columns P_{ftr}, R_{ftr}, and F_{ftr} list the `precision`, `recall`, and `F-score` of classifiers trained with the features proposed by ICON. Finally, columns P_{Δ}, R_{Δ}, and F_{Δ} list the improvement factors in `precision`, `recall`, and `F-score` respectively.

For our evaluation we used following well known classifiers: AdaBoost (or adaptive boost-

Table 4.3: Evaluation Results (Inference)

API	Mtds	Sen	Sen_C	Spec_{ICON}	Acc(%)
java.io	662	2417	78	56	71.8
Amazon S3 REST	51	1492	12	7	58.3
Paypal REST	33	151	20	14	70.0
Total	746	4060	110	77	70.0*

* Average; Mtds: Total no. of Methods; Sen: Total no. of Sentences; Sen_C: Total no. of constraint Sentences; Acc: Accuracy Spec_{ICON}: Total no. of temporal constraints correctly identified by ICON;

ing), Naïve Bayes, Winnow, Balanced Winnow, Decision Tree, Max Entropy, and c45. We used Naïve Bayes as the weaker classifier with AdaBoost. We used `Mallet` [mal] implementation of these classifiers for our experiments.

Our results show that ICON effectively identifies constraint sentences with the average (across different classifiers) `precision`, `recall`, and `F-score` of 79.0%, 60.0%, and 65.0%, respectively. Balanced Winnow performed best, with an average `precision` and `recall` of 78% and 77% respectively. Furthermore, our results also show the features proposed by ICON improves the precision of classification algorithms by an average of 18%. Our results also show a slight increase in the recall (average 1% gain).

RQ2: Accuracy in Inferring Temporal Constraints

Table 4.3 shows the effectiveness of ICON in inferring temporal constraints from the identified constraint sentences. Column "API" lists the names of the subject API. Columns "Mtds" and "Sen" list the number of methods and sentences in each subject API's. Column "Sen_C" lists the number of manually identified constraint sentences. Column "Spec_{ICON}" lists the number of sentences with correctly inferred temporal constraints by ICON. Column "Acc(%)" list percentage values of accuracy. Our results show that, out of 90 manually identified constraint sentences, ICON correctly infers temporal constraints with the average accuracy of 70.0%.

We next present an example to illustrate how ICON incorrectly infers temporal constraints from a constraint sentence. Consider the sentence "*if the stream does not support seek, or if this input stream has been closed by invoking its close method, or an I/O error occurs.*" from

`skip` method of `java.io.FilterInputStream` class. Although ICON correctly infers that method `close` cannot be called before current method, ICON incorrectly associates the phrase "support seek" with method `markSupported` in the class. The faulty association happens due to incorrect parsing of the sentence by the underlying NLP infrastructure. Such issues will be alleviated as the underlying NLP infrastructure improves.

We next present an example to illustrate how ICON fails to infer constraints from a constraint sentence. For instance, consider the sentence "*This implementation of the PUT operation creates a copy of an object that is already stored in Amazon S3.*" from `PUT Object-Copy` method description in `Amazon S3 REST` API. The sentence describes the constraint that the object must already be stored (invocation of `PUT Object`) before calling the current method. However, ICON cannot make the connection due to the limitation of the semantic graphs that do not list "already stored" as a "valid operation" on object. In the future, we plan to investigate techniques to further improve knowledge graphs to infer such implicit constraints.

RQ3: Comparison to Typed-Enforced Constraints

In this section, we compare the temporal constraints inferred from the natural language API descriptions to those enforced by the type-system (referred to as type-enforced constraint). The constraints that are enforced by the type-system can be enforced by IDEs. Hence, for such types of constraints, we do not require sophisticated techniques like ICON. For `java.io`, we define a type-enforced constraint as a constraint that mandates a method M accepting input parameter I of type T to be invoked after (at least one) a method M' whose return value is of type T. Since there are no types in REST APIs, for `Amazon S3`, we consider a constraint as a type-enforced constraint if the constraint is implicit in the `CRUD` semantic followed by REST operations. `CRUD` stands for resource manipulation semantic sequence create, retrieve, update, and delete. In particular, we consider a constraint as a type-enforced constraint, if the constraint mandates a DELETE, GET, or PUT operation on a resource to be invoked after a POST operation on the same resource.

To address this question, we manually inspected each of the constraints reported by ICON and classify it as a type-enforced constraint or a non type-enforced constraint. We observed that none of the constraints inferred by ICON from natural language text were classified as a type-enforced constraint. Hence, the constraints detected by ICON are not trivial enough to be enforced by a type system.

4.4.4 Summary

In summary, we demonstrate that ICON effectively identifies constraint sentences (from over 4000 API sentences) with the average precision, recall, and F-score of 79.0%, 60%, and 65% respectively [RQ1]. We also show that ICON infers temporal constraints from the constraint sentences an average accuracy of 70% [RQ2]. Furthermore, also provide discussion on why ICON does not or incorrectly infers temporal constraints. Finally, we provide a comparison of the temporal constraints inferred from natural language description against the temporal constraints enforced by a type system [RQ3].

4.4.5 Threats to Validity

Threats to external validity primarily include the degree to which the subject documents used in our evaluation are representative of true practice. To minimize the threat, we used API documents of three different APIs: JDK `java.io`, `Amazon S3 REST` API, `PayPal Payment REST` API. On one hand, Java is a widely used programming language and `java.io` is one of the main packages. In contrast, `Amazon S3 REST` API provides HTTP based access to online storage allowing developers the freedom to write clients applications in any programming language. Finally, `PayPal Payment REST` API provides a REST support for online financial transactions. The difference in the functionality provided by the three APIs 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 APIs.

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

4.5 Limitations and Future work

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

Validation of Method Descriptions. API documents can sometimes be misleading [TMTL12b,

RGL10], thus causing developers to write faulty client code. In the future, we plan to extend our approach to find documentation-implementation inconsistencies.

Inferring Implicit Constraints. The presented approach only infers temporal constraints explicitly described in the method descriptions. However, there are instances where the constraints are implicit. For instance, consider the method description for `markSupported` method in `BufferInputStream` class in Java, that states "*Test if this input stream supports* `mark`". Although trivial for a human to interpret that the method `markSupported` must be invoked before the method `mark`, our approach is unable to infer such implicit temporal constraints. In future work, we plan to investigate techniques to infer these implicit temporal constraints.

Extending Generic Dictionaries. The use of generic dictionaries for software engineering related text is sometimes inadequate. For instance, Wordnet matches "has" as a synonym for the word "get". Although valid for generic English, such instances cause our approach to incorrectly distinguish a constraint sentence from a regular sentence, or vice versa. In future work, we plan to investigate techniques to extend generic dictionaries for software engineering related text. In particular, Yang and Tan [YT13] recently proposed a technique for inferring semantically similar words from software context to facilitate code search. We plan to explore such techniques and evaluate the overall effectiveness of our approach after augmenting it with such techniques.

4.6 Chapter Summary

Despite being highly desirable, formal temporal constraints are absent in most APIs. In contrast, documentation of API methods contains detailed specifications of temporal constraints in natural language text. Manually writing formal specifications based on natural language text in API documents is prohibitively time-consuming and error-prone. To address this issue, we have proposed a novel approach called ICON to infer temporal constraints from natural language text of API documents. We used ICON to infer temporal constraints from the `PayPal Payment REST` API, the `Amazon S3 REST` API, and the commonly used package `java.io` in the JDK API. Our evaluation results show that ICON effectively identifies sentences describing temporal constraints with an average 79% precision and 60% recall, from more than 4000 sentences in subject API documents. Furthermore, ICON also achieves an accuracy of 70% in inferring 77 formal temporal constraints from these temporal constraint sentences.

CHAPTER 5

WHYPER: Towards Automating Risk Assessment of Mobile Applications

This chapter introduces WHYPER: a framework using Natural Language Processing (NLP) techniques to identify sentences that describe the need for a given permission in an application description. The remainder of this Chapter proceeds as follows. Section 5.1 describes the motivation behinf WHYPER. Section 5.3 presents the overview of the WHYPER framework. Section 5.4 presents our framework and implementation. Section 5.5 presents evaluation of our framework.

5.1 Introduction

Application markets such as Apple's App Store and Google's Play Store have become the *de facto* mechanism of delivering software to consumer smartphones and mobile devices. Markets have enabled a vibrant software ecosystem that benefits both consumers and developers. Markets provide a central location for users to discover, purchase, download, and install software with only a few clicks within on-device market interfaces. Simultaneously, they also provide a mechanism for developers to advertise, sell, and distribute their applications. Unfortunately, these characteristics also provide an easy distribution mechanism for developers with malicious

intent to distribute malware.

To address market-security issues, the two predominant smartphone platforms (Apple and Google) use starkly contrasting approaches. On one hand, Apple forces all applications submitted to its App Store to undergo some level of manual inspection and analysis before they are published. This manual intervention allows an Apple employee to read an application's description and determine whether the different information and resources used by the application are appropriate. On the other hand, Google performs no such checking before publishing an application. While Bouncer [Loc12] provides static and dynamic malware analysis of published applications, Google primarily relies on permissions for security. Application developers must request permissions to access security and privacy sensitive information and resources. This permission list is presented to the user at the time of installation with the implicit assumption that the user is able to determine whether the listed permissions are appropriate.

However, it is non-trivial to classify an application as malicious, privacy infringing, or benign. Previous work has looked at permissions [FFC+11, ZWZJ12, PGS+12, CRTE13], code [EKKV11, EOMC11, ZJ12, GZZ+12, GCEC12], and runtime behavior [EGC+10, HHJ+11, YY12]. 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 chapter, 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.

This chapter makes the following main contributions:

- We propose the use of NLP techniques to help bridge the semantic gap between what mobile applications do and what users expect them to do. To the best of our knowledge, this work is the first attempt to automate this inference.
- We evaluate our framework on 581 popular Android application descriptions containing nearly 10,000 natural-language sentences. Our evaluation demonstrates substantial improvement over a basic keyword-based searching.
- We provide a publicly available prototype implementation of our approach on the project website [proc].

WHYPER is merely the first step in bridging the semantic gap of user expectations. There are many ways in which we see this work developing. Application descriptions are only one form of input. We foresee possibility in also incorporating the application name, user reviews, and potentially even screen-shots. Furthermore, permissions could potentially be replaced with specific API calls, or even the results of dynamic analysis. We also see great potential in developing automatic or partially manual techniques of creating and fine-tuning permission semantic models.

Finally, this work dovetails nicely with recent discourse concerning the appropriateness of Android permissions to protect user security [EOM09, BKvOS10, FGW11, FHE+12]. The overwhelming evidence indicates that most users do not understand what permissions mean,

even if they are inclined to look at the permission list [FHE+12]. On the other hand, permission lists provide a necessary foundation for security. Markets cannot simultaneously cater to the security and privacy requirements of all users [ME10], and permission lists allow researchers and expert users to become "whistle blowers" for security and privacy concerns [EGC+10]. In fact, a recent comparison [HYG+13] of the Android and iOS versions of applications showed that iOS applications overwhelmingly more frequently use privacy-sensitive APIs. Tools such as WHYPER can help raise awareness of security and privacy problems and lower the sophistication required for concerned users to take control of their devices.

5.2 Android Applications

We now present a brief background on Android Applications. Google Play is a digital application distribution service operated by Google. Applications can be installed from the device or the Google Play website. Google Play can update the applications the user selects automatically, or users can update them on a per-case basis or update all applications at once.

Google currently uses an in-house automated anti-virus system to remove malicious applications uploaded on to the marketplace code named Bouncer [Loc12]. Bouncer is meant to prevent repeat-offender developers, as well as check for anomalies in uploaded apps [1]. a recent study conducted by Kaspersky Labs [kas] indicate a three-fold increase in the number of malicious programs in just the second financial quarter of 2012.

5.3 WHYPER Overview

We next present a brief overview of the WHYPER framework. The name WHYPER itself is a word-play on phrase *why per*missions. We envision WHYPER to operate between the application market and end users, either as a part of the application market or a standalone system as shown in Figure 5.1.

The primary goal of the WHYPER framework is to bridge the semantic gap of user expectations by determining why an application requires a permission. In particular, we use application descriptions to get this information. Thus, the WHYPER framework operates between the application market and end users. Furthermore, our framework could also serve to help devel-

[1] Bouncer is credited with reducing malware by 40% between the first and second quarters of 2011.

opers with the feedback to improve their applications, as shown by the dotted arrows between developers and WHYPER in Figure 5.1.

A straightforward way of realizing the WHYPER framework is to perform a keyword-based search on application descriptions to annotate sentences describing sensitive operations pertinent to a permission. However, we demonstrate in our evaluation that such an approach is limited by producing many false positives. We propose Natural Language Processing (NLP) as a means to alleviate the shortcomings of keyword-based searching. In particular, we address the following limitations of keyword-based searching:

1. **Confounding Effects**. Certain keywords such as "contact" have a confounding meaning. For instance, *'... displays user contacts, ...'* vs *'... contact me at abc@xyz.com'*. The first sentence fragment refers to a sensitive operation while the second fragment does not. However, both fragments include the keyword "contact".

   To address this limitation, we propose NLP as a means to infer semantics such as whether the word refers to a resource or a generic action.

2. **Semantic Inference**. Sentences often describe a sensitive operation such as *reading contacts* without actually referring to keyword "contact". For instance, *"share... with your friends via email, sms"*. The sentence fragment describes the need for *reading contacts*; however the "contact" keyword is not used.

   To address this limitation, we propose to use API documents as a source of semantic information for identifying actions and resources related to a sensitive operation.

To the best of our knowledge, ours is the first framework in this direction. We next present the key NLP techniques used in this work. We next describe the threat model that we considered while designing our WHYPER framework.

5.3.1 Use Cases and Threat Model

WHYPER is an enabling technology for a number of use cases. In its simplest form, WHYPER could enable an enhanced user experience for installing applications. For example, the market interface could highlight the sentences that correspond to a specific permission, or raise warnings when it cannot find any sentence for a permission. WHYPER could also be used by market providers to help force developers to disclose functionality to users. In its primitive

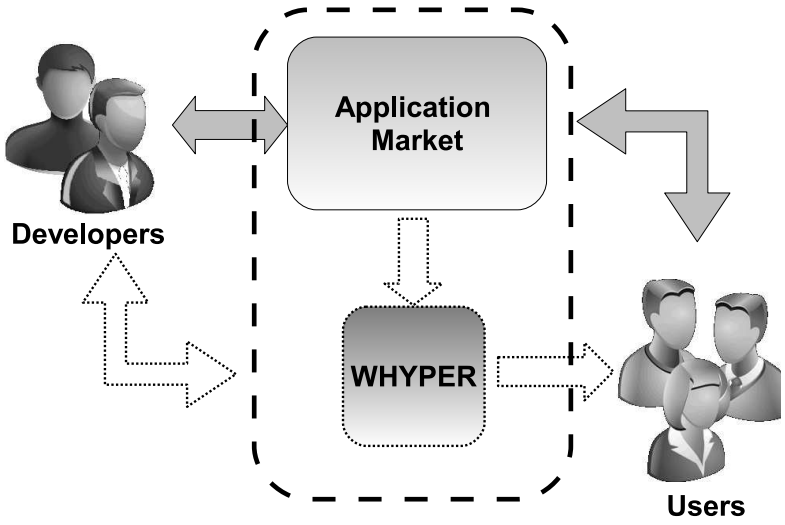

Figure 5.1: Overview of WHYPER

form, market providers could use WHYPER to ensure permission requests have justifications in the description. More advanced versions of WHYPER could also incorporate the results of static and dynamic application analysis to ensure more semantically appropriate justifications. Such requirements could be placed on all new applications, or iteratively applied to existing applications by automatically emailing developers of applications with unjustified permissions. Alternatively, market providers and security researchers could use WHYPER to help triage markets [CRTE13] for dangerous and privacy infringing applications. Finally, WHYPER could be used in concert with existing crowd-sourcing techniques [LAH+12] designed to assess user expectations of application functionality. All of these use cases have unique threat models.

For the purposes of this chapter, we consider the generic use scenario where a human is notified by WHYPER if specific permissions requested by an application are not justified by the application's textual description. WHYPER is primarily designed to help identify privacy infringements in relatively benign applications. However, WHYPER can also help highlight malware that attempts to sneak past consumers by adding additional permissions (e.g., to send premium-rate SMS messages). Clearly, a developer can lie when writing the application's description. WHYPER does not attempt to detect such lies. Instead, we assume that statements describing unexpected functionality will appear out-of-place for consumers reading an application description before installing it. We note that malware may still hide malicious functionality (e.g., eavesdropping) within an application designed to use the corresponding permission

(e.g., an application to take voice notes). However, false claims in an application's description can provide justification for removal from the market or potentially even criminal prosecution. WHYPER does, however, provide defense against developers that simply include a list of keywords in the application description (e.g., to aid search rankings).

Fundamentally, WHYPER identifies if there is a possible implementation need for a permission based on the application's description. In a platform such as Android, there are many ways to accomplish the same implementation goals. Some implementation options require permissions, while others do not. For example, an application can make a call that starts Android's address book application to allow the user to select a contact (requiring no permission), or it can access the address book directly (requiring a permission). From WHYPER's perspective, it only matters that privileged functionality may work expectedly when using the application, and that functionality is disclosed in the application's description. Other techniques (e.g., Stowaway [FCH+11]) can be used to determine if an application's code actually requires a permission.

5.4 WHYPER Design

We next present our framework for annotating the sentences that describe the needs for permissions in application descriptions. Figure 5.2 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.

5.4.1 Preprocessor

The preprocessor accepts natural-language application descriptions and preprocesses the sentences, to be further analyzed by the intermediate-representation generator. The preprocessor

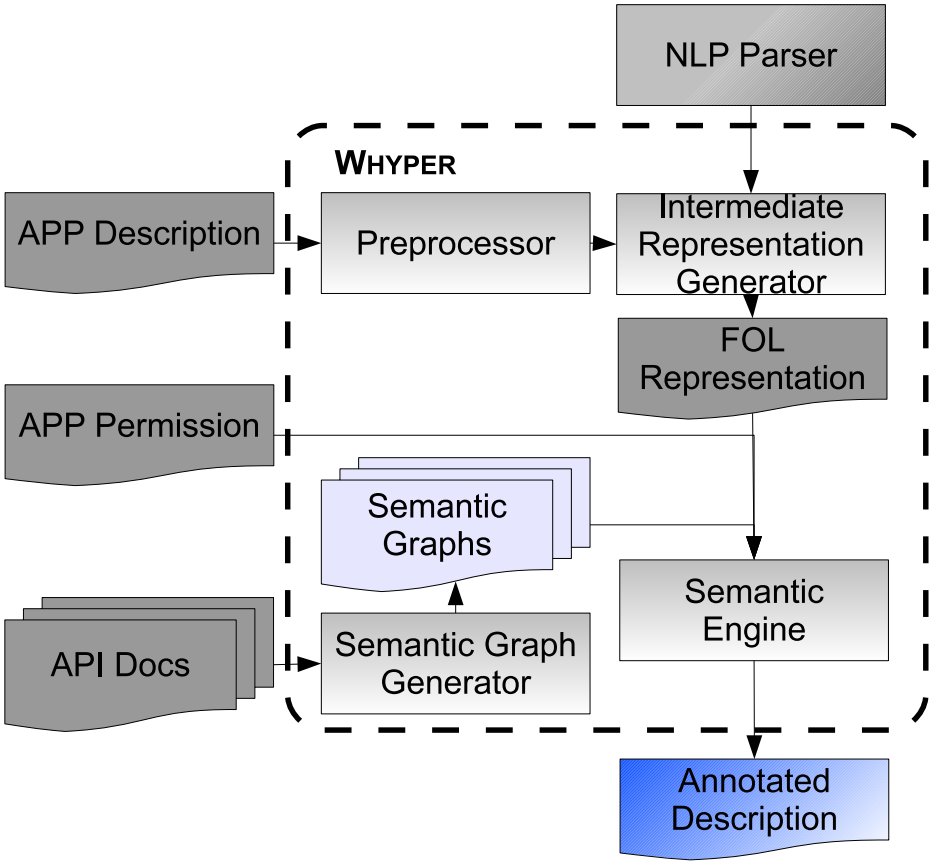

Figure 5.2: Overview of WHYPER framework

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 [ea98] 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 [proc].
- 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.

5.4.2 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 [KM03a]. 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:** [dMMM06, dMM08] 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 5.3 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 [dMMM06, dMM08].

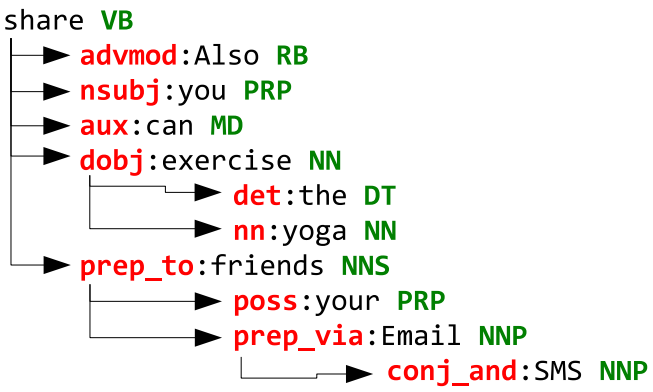

Figure 5.3: Sentence annotated with Stanford dependencies

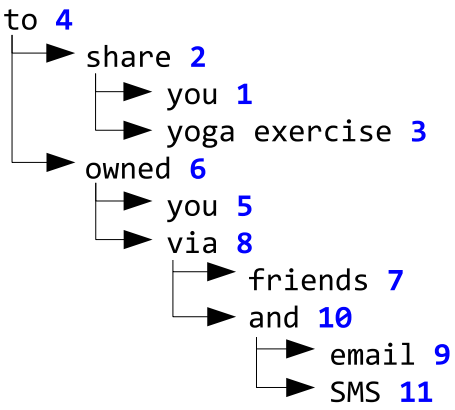

Figure 5.4: First-order logic representation of annotated sentence in Figure 5.3

5.4.3 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 [SPKB09, SSP10, PXZ+12]. 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 [Bog00] 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 [dMMM06, dMM08, KM03a, KM03c]. 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 [Bog00, ST97, SPKB09, Gre99, PXZ+12] 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 5.1 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 [proc]. Figure 5.4 shows the FOL representation of the sentence in Figure 5.3. 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 5.4, 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 5.3).

The FOL representation helps us effectively deal with the problem of *confounding effects* of keywords as described in Section 5.3. 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.

Table 5.1: Examples of 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

5.4.4 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 5.5 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 5.4.5 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 4. 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 [DRS+09] 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 [DRS+09] 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.

Algorithm 4 Sentence_Annotator

Require: K_Graph g, FOL_rep rep, Boolean $recursion$
Ensure: 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
```

In the context of the FOL representation shown in Figure 5.4, we invoke Algorithm 4 with the semantic graph shown in Figure 5.5. 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 5.4). Our algorithm then explores the preceding predicates and finds predicate "share" (annotated 2 in Figure 5.4). 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.

5.4.5 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

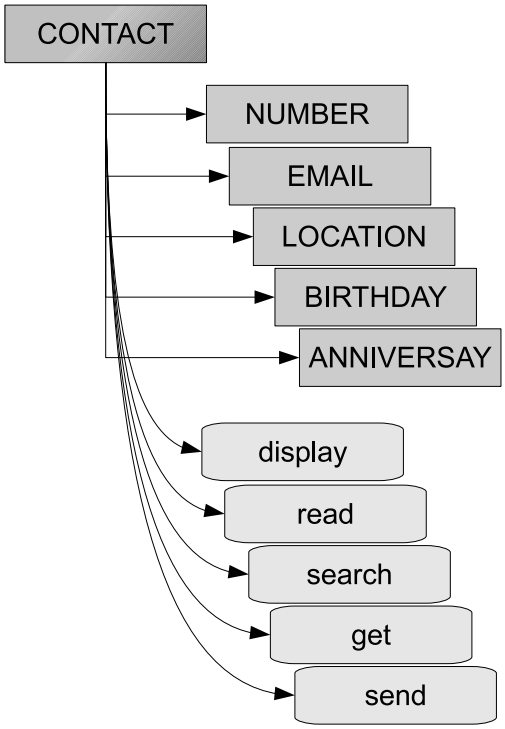

Figure 5.5: Semantic Graph for the READ_CONTACT permission

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 [AZHL12] 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`[2] class that is associated with READ_CONTACT permission. Third, we systematically inspect the member variables and member methods to identify actions and subordinate

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

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, `Contracts--Contract.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 5.5 shows a sub-graph of graph for READ_CONTACT permission.

5.5 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 ?

5.5.1 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).

5.5.2 Evaluation Setup

We first manually annotated the sentences in the application descriptions. We had three researchers [3] independently annotate sentences in our corpus, ensuring that each sentence was annotated by at least two researchers. The individual annotations were then discussed by all three researchers to reach to a consensus. In our evaluation, we annotated a sentence as a permission sentence if at least two researchers agreed that the sentence described the need for a permission. Otherwise we annotated the sentence as a permission-irrelevant sentence.

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.

[3] Rahul Pandita, Xusheng Xiao, Wei Yang

Table 5.2: 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.

4. FN: A sentence that WHYPER incorrectly identifies as not a permission sentence.

Table 5.2 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 researchers. 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 5.4 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.

Table 5.3: 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

In statistical classification [Ols08], *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:

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

5.5.3 Results

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

RQ1: Effectiveness in identifying permission sentences

In this section, we quantify the effectiveness of WHYPER in identifying permission sentences by answering RQ1. Table 5.3 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 [DRS+09] 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 synchroniza-*

tion 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 [KM03a] 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 negatives). 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.*" [4] 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 [KM03a] 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:

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

> 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.

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 5.5.2, 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 5.5 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, f-scores, and accuracy respectively. Our results show that, in comparison to keyword-based searching, WHYPER effectively identifies permission sentences

Table 5.4: 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

Table 5.5: 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

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 5.4. 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 5.5). 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 pro-

duced by our framework. As described in Section 5.5.3 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 5.5 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.

5.5.4 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.

5.5.5 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 researchers. The results of our experiments are publicly available on the project website [proc].

5.6 Discussions and Future work

Our framework currently only takes into account application descriptions and Android API documents to highlight permission sentences. Thus, our framework can semi-formally enumerate the uses of a permission. This information can be leveraged in the future to enhance searching for desired applications.

Furthermore, the outputs from our framework could be used in conjunction with program analysis techniques to facilitate effective code reuse. For instance, our framework outputs the reasons of why a permission is needed for an application. These reasons are usually the functionalities provided by the application. In future work, we plan to locate the code fragments that implement the described functionalities. Such mapping can be used as indexes to existing code searching approaches [TX07, Rei09] to facilitate effective reuse.

Modular Applications: In our evaluations, we encountered cases where a description referring to another application where the permission sentences were described. For instance, consider the following description sentence *"Navigation2GO is an application to easily launch Google Maps Navigation."*.

The description states that the current application will launch another application "Google Maps Navigation", and thus requires the permissions required by that application. Currently, our framework does not deal with such cases. We plan to implement deeper semantic analysis of description sentences to identify such permission sentences.

Generalization to Other Permissions: WHYPER is designed to identify the textual justification for permissions that protect "user understandable" information and resources. That is, the permission must protect an information source or resource that is in the domain of knowl-

edge of general smartphone users, as opposed to a low-level API only known to developers. The permissions studied in this chapter (i.e., address book, calendar, microphone) fall within this domain. Based on our studies, we expect similar permissions, such as those that protect SMS interfaces and data stores, the ability to make and receive phone calls, read call logs and browser history, operate and administer Bluetooth and NFC, and access and use phone accounts will have similar success with WHYPER.

Due to current developer trends and practices, there is class of permissions that we expect will raise alarms for many applications when evaluated with WHYPER. Recent work [EGC+10, EOMC11] has shown that many applications leak geographic location and phone identifiers without the users knowledge. We recommend that deployments of WHYPER first focus on other permissions to better gauge and account for developer response. Once general deployment experience with WHYPER has been gained, these more contentious permissions should be tackled. We believe that adding justification for access to geographic location and phone identifiers in the application's textual description will benefit users. For example, if an application uses location for ads or analytics, the developer should state this in the application description.

Finally, there are some permissions that are implicitly used by applications and therefore will have poor results with WHYPER. In particular, we do not expect the Internet permission to work well with WHYPER. Nearly all smartphone applications access the Internet, and we expect attempts to build a semantic graph for the Internet permission will be largely ineffective.

Results Presentation: A potential after-effect of using WHYPER on existing application descriptions might be more verbose application descriptions. One can argue that it would lead to additional burden on end users to read a lengthy description. However, such additional description provides an opportunity for the users to make informed decisions instead of making assumptions. In future work, we plan to implement and evaluate interfaces to present users with information in a more efficient way, countering user fatigue in case of lengthy descriptions. For example, we may consider using icons in different colors to represent permissions with and without explanation.

5.7 Chapter Summary

In this chapter, I have presented WHYPER, a framework that uses Natural Language Processing (NLP) techniques to determine why an application uses a permission. We evaluated our prototype implementation of WHYPER on real-world application descriptions that involve three

permissions (address book, calendar, and record audio). These are frequently-used permissions that protect privacy and security sensitive resources. Our evaluation results show that WHYPER achieves an average precision of 82.8%, and an average recall of 81.5% for three permissions. In summary, our results demonstrate great promise in using NLP techniques to bridge the semantic gap of user expectations to aid the risk assessment of mobile applications.

CHAPTER 6

Discovering Likely Mappings Between APIs using Text Mining

Developers often migrate applications to release different versions for supporting application programming interfaces (APIs) of various platforms/programming-languages. To migrate an application written using one API (source) to another API (target), a developer needs to know how the methods in the source API map to the methods in the target API. This chapter introduces TMAP [1]: ***T**ext **M**ining based approach to discover likely **API** mappings* using the similarity in the textual description of the source and target API documents The remainder of this chapter is organized as follows. Section 6.1 introduces the approach. Section 6.2 presents a real world example that motivates our approach. Section 6.3 presents TMAP approach. Section 6.4 presents evaluation of TMAP. Section 6.5 presents a brief discussion and future work.

6.1 Introduction

Software is ubiquitous and of late people are increasingly interacting with software applications that run on variety of software platforms on daily basis. To retain existing users (and attract new users) across different platforms, developers are increasingly releasing different

[1] This work was done in collaboration with Raoul Jetley, Sithu Sudarsan, and Laurie Williams

versions of their applications. For example, a typical mobile software developer often releases his/her applications on all the popular mobile platforms, such as Android, iOS, and Windows, which often involves rewriting applications in different languages. For instance, Java is preferred language for implementing Android applications and Objective-C for iOS application. In context of desktop software, many well-known projects, such as JUnit, Hibernate provide multiple versions in different programming languages, to attract developer community to use these libraries across those languages.

To assist developers with software migration there are existing language migration tools, such as Java2CSTranslator [jav]. However, such tools require a programmer to manually input how methods in a source language's Application Programming Interfaces (API) maps to the methods of the target language's API. Given a typical language (or platform) exposes a large number of API methods for developers to reuse, manually writing these mappings is prohibitively resource intensive and may be error prone.

The goal of this research is to support software developers in migrating an application from a source API to a target API by automatically discovering likely method mappings across APIs using text mining on the natural language API method descriptions

Existing approaches in literature address the problem of finding method mapping between APIs using static [ZTX+10] and dynamic [GGP13] analysis. Recently Nguyen et al. [NNNN14] further proposed to apply statistical language translation techniques to achieve language migration by mining large corpora of open source software repositories. However, these approaches require as an input manually ported (or at least functionally similar) software across source and target APIs. Since static analysis and mining approaches [ZTX+10, NNNN14] leverage source code analysis, accuracy of such approaches is dependent on the quality of the code under consideration. Likewise, accuracy of dynamic approaches [GGP13] is dependent on the quality and completeness of test inputs to dynamically execute the API behavior comprehensively.

To address the shortcomings of existing program-analysis based approaches, we propose to use the natural language API method descriptions to discover the method mappings across APIs. Our intuition is: *since the documentation is targeted towards developers, there may be an overlap in the language used to describe similar concepts that can be leveraged.* In general, 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. A method description typically outlines specifications in terms of the expectations of the method arguments and

functionality of method in general.

This chapter presents TMAP: An approach that leverages the natural language method descriptions to discover the likely method mapping between APIs. TMAP stands for ***T**ext **M**ining based approach to discover likely **API** method mappings*. In particular, TMAP proposes to create a vector space model [Sin01, MRS08] of the target API method descriptions. TMAP then queries the vector space model of target API using automatically generated queries from the source API method descriptions. TMAP automates the query generation in source API using the concepts from text mining, such as emphasizing (or omitting) certain keywords over others and querying multiple facets (such as class description, package names, and method description).

We pose the following research question: *How accurately can the similarity in the language of API method descriptions be leveraged to discover likely API Mappings?* To answer our question, we apply TMAP to discover likely API mappings for 15 classes across: 1) `Java` and `C#` API; 2) `Java ME` [2] and `Android` API. We also compare the discovered mappings with two state-of-the-art static and dynamic analysis based approaches: Rosetta [GGP13] and StaMiner [NNNN14]. Our results indicate that TMAP on average found relevant mappings for 57% more methods compared to Rosetta and StaMiner. Furthermore, our results also indicate that TMAP found on average exact mappings for 6.5 more methods per class with a maximum of 21 additional exact mappings for a single class as compared to previous approaches.

In summary, TMAP leverages natural language description of APIs to discover likely mapping thus facilitating cross API migration of applications. Since TMAP analyzes API documents in natural language, the proposed approach is reusable, independent of the programming language of the library. This chapter makes the following major contributions:

- A text mining based approach that effectively discovers mapping between source and target API.
- A prototype implementation of our approach based on extending the Apache Lucene [luc]. An open source implementation of our prototype can be found at our website. [prob]
- An evaluation of our approach on 5 classes in Java ME to Android API and 10 Classes Java to C# API. The evaluation results and artifacts are publicly available on the project website.

[2] Java Platform Micro Edition

javax.microedition.lcdui Class Graphics drawString

`public void drawString(String str,int x,int y,int anchor)`

Draws the specified String using the current font and color. The x,y position is the position of the anchor point. See anchor points.

Parameters

str - the String to be drawn

x - the x coordinate of the anchor point

y - the y coordinate of the anchor point

anchor - the anchor point for positioning the text

Throws

NullPointerException - if str is null

IllegalArgumentException - if anchor is not a legal value

See Also

`drawChars(char[], int, int, int, int, int)`

Figure 6.1: `drawString` API method description from `Graphics` class in the Java ME API

6.2 Example

We next present an example to motivate our work and list the considerations for applying text mining techniques on API documents. The example is from the real Java Platform Micro Edition (Java ME) formerly known as J2ME and the Android API. Both Java ME and Android use Java as the language of implementation and are targeted towards hand-held devices. However, our approach is independent of language of implementation and thus can be used for any source and target API.

Figure 6.1 shows the API method description of `drawString` method from `javax.micro--edition.lcdui.Graphics` class in Java ME API. Figure 6.2 shows the API method description of method `drawText` method from `android.graphics.Canvas` class in Android API.

Notice the overlap in the language of these two methods. From a human perspective, it is easy to conclude that the two methods offer similar functionality. However, the Android API has more than 23,000 public methods. Manually going through each method description to find similar methods is prohibitively time consuming, supporting the need for automation. A naive solution to automate the task is to perform keyword-based search.

To demonstrate the difficulties faced by keyword-based search in above example, we searched

android.graphics Class Canvas drawText

`public void drawText (String text, float x, float y, Paint paint)`

Draw the text, with origin at (x,y), using the specified paint. The origin is interpreted based on the Align setting in the paint.

Parameters

text - The text to be drawn

x - The x-coordinate of the origin of the text being drawn

y - The y-coordinate of the origin of the text being drawn

paint - The paint used for the text (e.g. color, size, style)

Figure 6.2: `drawText` API method description from `Canvas` class in the Android API

the Android API description with the keywords listed in Table 6.1 using the Apache Lucene [luc] framework. Lucene is a high-performance, full-featured text search engine library written entirely in Java. The column "Query" describes the keywords we used to perform keyword-based search, The column "Hits" lists the number of matches found, and The column "Top-10" lists the rank of the first relevant method in top-ten results. For instance, when we searched for the class name "Graphics" in the Android API, we did not get any results. We also did not get any results for when we searched for method name using keywords "drawString".

When we searched the Android API using the words in the method signature as keywords, we got a 23,547 results (almost all methods in Android API). The high number of results is because of the confounding effects of keywords, such as "public". Most of the methods signatures have the keyword "public". Although the ranking mechanisms in Lucene did rearrange the results moving methods with most keyword matches first, we did not find a relevant method in top-ten results. Likewise, the words in method descriptions as a whole or in parts also did not yield better results. In the example, a combination of various attributes, such as method name split in camel case notation "draw String", keywords from both class and method description resulted in the Android API equivalent method `drawText` shown in Figure 6.2 in the top ten results.

The example demonstrates difficulties faced by simple keyword-based searches and text mining in general to discover likely method mappings. The TMAP approach in general addresses the following difficulties in applying text mining approaches on natural language API method descriptions:

Table 6.1: Query Results

#	Query	Hits	Top-10
1	Class Name: "Graphics"	0	-
2	Method Name: "drawString"	0	-
3	Method Signature	23547	-
4	Method Description: (Complete)	16820	
5	Method Description: (summary sentences)	94230	-
6	Combination	1479	**3**

'-'=No Match in Top-10 results.

1. *Confounding effects*. Certain method names have a confounding effects. For instance, `toString()`,`get()`,`set()` are too generic and tend to have similar descriptions across different method definitions. These generic methods often cause interference with the output of text mining based approaches. The problem is to automatically identify such methods to de-emphasize their importance in a query.

2. *Weights*. Not all terms in a method descriptions are equally important keywords. For instance, the term "zip" in the sentence "opens a zip file" is more important than the terms "opens" and "files" as the term empathizes on the specific type of file. The problem is to automatically identify the importance of a term.

3. *Structure*. API documents are not flat contiguous text blobs. They have well defined structure, that is often shared by method descriptions. Ignoring the structure may cause ineffective queries negatively affecting the results. The problem is to effectively aggregate the results of querying individual API document elements (such as class description, class names, method names, method descriptions).

6.3 Approach

We next present our approach for discovering likely mappings of API methods across APIs. Figure 6.3 provides an overview of the ICON approach. The ICON approach consists of three major components: 1) an Indexer; 2) Query Builder; and 3) Searcher components.

The Indexer accepts the API documents of the target API and creates indexes (a vector

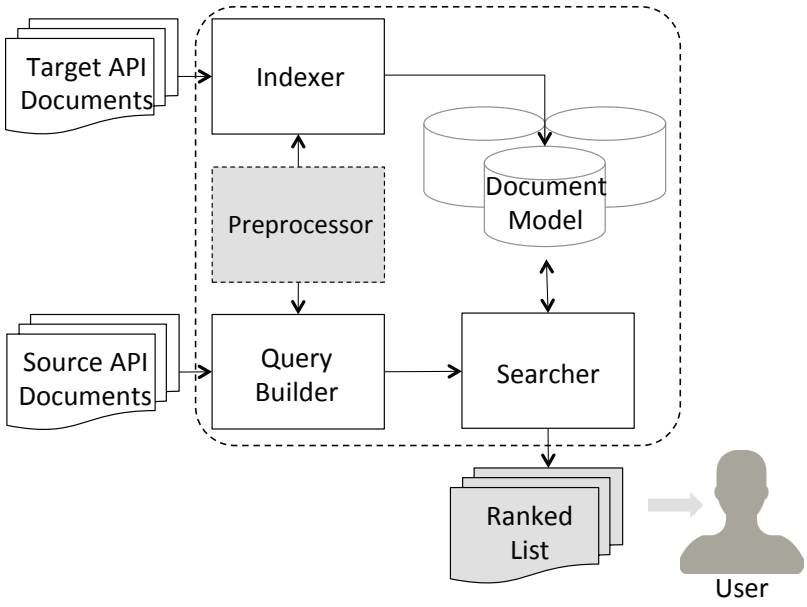

Figure 6.3: Overview of ICON approach

space model) of these documents by extracting intermediate contents from the method descriptions. The Query Builder accepts the API documents of the source API and creates queries to be executed on the indexes (a vector space model). Finally, Searcher component execute the queries on the indexes and generates an ordered set of the API methods from target API documents as mapping results to be presented to the developers for confirmation. We next describe each component in detail.

6.3.1 Indexer

This component accepts the API method descriptions of the target API and creates indexes (a vector space model) of these documents. In particular, Indexer extracts the following fields from the API method descriptions:

- **F1) Type Name**: The name of enclosing class/interface of the method. For the method description of `drawString` shown in Figure 6.1, indexer extracts the type Name as "Graphics".
- **F2) Package Name**: The package name of the enclosing type. For the method description

shown in Figure 6.1, indexer extracts the package name as “javax.microedition.lcdui”.

- **F3) Method Name**: The name of the method. For the method description shown in Figure 6.1, indexer extracts the method name as “drawString”.
- **F4) Method Modifiers**: The access modifiers of the method. For the method description shown in Figure 6.1, indexer extracts the method modifiers as “public”.
- **F5) Method Return Type**: The return type of the method. For the method description shown in Figure 6.1, indexer extracts the return type as “void”.
- **F6) Exceptions Thrown**: Name of the exceptions thrown by the method. For the method description shown in Figure 6.1, indexer extracts the exception names as “NullPointerException” and “IllegalArgumentException”.
- **F7) Parameters**: The name and type information of parameters. For the method description shown in Figure 6.1, indexer extracts the parameter name and type information as “String str”,“int x”,“int y”, and “int anchor”.
- **F8) Class Description**: The description of enclosing type. Class description is not shown in Figure 6.1 for the space considerations.
- **F9) Method Description**: The description of the method. For the method description shown in Figure 6.1, indexer extracts the method description as the all the text except method deceleration in Line 1.

This step is required to extract the desired descriptive text from the API method descriptions. Getting structured descriptions facilitates searching on individual categories. Thus step allows TMAP to deal with the *structure* issue as presented in Section 6.2. Different API documents may have different styles of presenting information to developers. Such stylistic differences may also include the difference in the level of the details presented to developers. TMAP relies only on basic fields that are generally available for API methods across different presentation styles.

After extracting the desired information, extracted text is further preprocessed. The preprocessing steps are required to make the text amenable to text mining techniques that are used in the subsequent phases of the TMAP approach. In particular, TMAP performs the following basic prepossessing steps:

- **P1) Presentation Elements**: A typical API method description is often interleaved with presentation elements for better readability. For instance, `JavaDoc` provides a list of identifiers such as `@Code` and `@link`. These identifiers are automatically translated into presentation markup, such as links and fonts. Although such elements are part of the description text, these elements often cause noise in the text mining techniques to compute relevance based on the query. Therefore, this preprocessing step cleans the method descriptions to remove such elements. We use a static list of presentation elements to achieve cleaning in this step with relatively high accuracy.

- **P2) Split Package Notation**: In method descriptions, the "." character is used as a separator character for package names like "`javax.microedition.lcdui`". We use regular expressions to split the package name into constituent words to facilitate search on individual words in the package name. For example, "`javax.microedition.lcdui`" is split into "javax microedition lcdui".

- **P3) Split CamelCase Notation**: API method descriptions are often interleaved with programming identifiers, such as class names and method names. Oftentimes these identifiers use CamelCase notation. The CamelCase notation is the *de facto* mechanism used by programmers for combining phrases into a single word, such that each word in the phrase begins with a capital letter. Previous research [LM07] demonstrated the benefit of splitting the CamelCase word into its constituent phrase for automated code completion. TMAP splits such identifiers into constituent phrases to better facilitate searching. TMAP leverages the well-formed structure of CamelCase notations to encode a regular expression to achieve splitting with relatively high accuracy. For example, "`drawString`" is split into "draw String".

- **P4) Lowercase**: This step involves converting the text description to lower case. The step is performed to normalize the text for making the keyword match case intensive, further increasing the range of queries.

- **P5) Stemming**: This steps transforms the words in the description to their base form. Stemming is very effective in extending the range of keyword based queries to match various operational forms of the words. For instance, "has", "have", and "had" are mapped to the stem "ha".

After preprocessing, TMAP next creates indexes for the API method descriptions. An index is collection of documents where each document is made up of values organized into well defined fields. TMAP considers every method description as an individual document and uses the following major fields (as previously described): 1) combination of package and class name; 2) class description; 3) method signature; 4) method name; and 5) method description. The values are of these fields is text after preprocessing. TMAP uses a vector space model representation of the documents for each field. Vector space model or term vector model is an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers and their frequency of occurrence. In case of TMAP, each word is considered as a term except the stop words, such as "a", "the", and "and".

6.3.2 Query Builder

This component accepts the API method descriptions of the source API and creates queries for method descriptions. These queries are executed on the target API index to retrieve an ordered list relevant API methods. In particular, Query Builder uses the same preprocessing steps followed by Indexer (listed in Section 6.3.1). After extracted desired descriptive text from the API method descriptions, this component systematically creates search queries to search for different fields in Indexes. Keywords for searching in "Type Name", "Type Description", "Method Name", and "Method Description" fields are derived from their equivalents in the extracted descriptive text.

For instance, consider the method description shown in Figure 6.4 and equivalent query in shown in Figure 6.5. The Keywords for "Type Name" are derived from preprocessing "java.util.Iterator" resulting in "`java util iter`". Notice the package notation is split into individual words and "Iterator" is further transformed to lower case and its stem "iter". Likewise, keywords for field "Method Name" is derived by preprocessing "hasNext", which is first split into "has Next" and then transformed using stemming into "`ha next`" (*ha* being stem of word *has*).

For generating keywords to query the "Type Description" field we consider following heuristic: *Heuristic H1: the first paragraph or the first five sentences of the type description (whichever is shorter) provides reasonable keywords for searching equivalent class in target API.*

Likewise, for generating the keywords to query the "Method Description" field, we consider the following heuristic: *Heuristic H2: the first paragraph or the first two sentences of the*

java.util Interface Iterator **hasNext**

`boolean hasNext()`

Returns `true` if the iteration has more elements. (In other words, returns true if `next()` would return an element rather than throwing an exception.)

Returns:

`true` if the iteration has more elements

Figure 6.4: API Method Description of `hasNext` method in `Iterator Interface` from Java API

Type Name: `java util iter`

Type Description: `iter over collect iter take enumer java collect framework`

Method Name: `ha next`

Method Description: `return true iter ha more element other word return true next would return element rather than throw except`

Figure 6.5: Query based on API method description of `hasNext` method in `Iterator Interface` from Java API

method description (whichever is shorter) provides reasonable keywords for searching equivalent method in target API.

TMAP uses these heuristics to improve the performance of searching infrastructure that tends to be inversely proportional to the complexity and length of the query. Using all the descriptive text as keywords results in a verbose query. As the number of keywords in a query increases the effectiveness of the query decreases. A large number of keywords have higher probability of matching large number of documents in comparison to a query with fewer keywords. In contrast, we observed that the document writers tend to describe the general overview of class and method description in the first few sentences followed by implementation and design specific details. We thus focused on the words in these overview sentences to create queries instead of using entire descriptive text.

Weights for terms. As mentioned in Section 6.2, all terms in a method description are not equally important keywords. TMAP further enhances the query by quantifying the importance of a term in the method description and use that as the weight of the corresponding keywords in the query. In particular, we propose to use tf-idf [MRS08] as a means to quantify importance of a term. For each term in the method description TMAP calculates the number of times the term occurs in that method description as $freq_{mtd}$. TMAP also calculates the maximum frequency of any term in the document as $freq_{MAX}$. TMAP then calculates the number of documents in the corpus that contains the term as $freq_{doc}$. TMAP finally calculates the tf-idf score of the term (as listed [MRS08]) as:

$$\texttt{tf-idf} = (0.5 \ + \frac{0.5 \ X \ freq_{mtd}}{freq_{MAX}}) \ * log(1 \ + \ \frac{total_{mtd}}{freq_{doc}})$$

The calculated tf-idf values of terms are normalized to a range of 0 to 1 (both 0 and 1 inclusive) for each document. The normalized tf-idf sore of the top-k term is then used as the weights for the corresponding keywords occurring in the query.

For the API method description shown in Figure 6.4, TMAP calculates “iter”, “has”, and “element” as most the important terms with normalized tf-idf scores of 1.0, 1.0, and 0,6 respectively. We augment the query shown in Figure 6.5 with the computed weights for the keywords respectively.

6.3.3 Searcher

The searcher component accepts the query from Query Builder component and queries the index generated by Indexer component. The results are then ranked and presented to the end

user for review. The searcher is realized as follows. First all the documents that match the keywords and clauses in a query are returned. Then, the returned documents are ranked using the cosine similarity [Sin01] of the terms in query and the terms in returned documents. In mathematics, Cosine similarity is a numerical statistic to measure the similarity between two vectors. In information theory [MRS08], cosine similarity is the standard statistic to rank relevant documents.

6.3.4 Implementation

We implemented a prototype version of the TMAP approach. We first manually download the HTML version of API documents of libraries. We then implemented a parser for extracting the requisite text from these documents using Jsoup [3], which is a java library for working with HTML documents. In particular our prototype implementation parses: 1) Oracle's Javadoc style; 2) Android style documentation; and 3) Microsoft's MSDN documentation.

We next implemented the indexing, query building, and searching infrastructure using the Apache Lucene [luc]. Lucene is a high-performance, full-featured text search engine library written entirely in Java. Our prototype implementation and evaluation subjects are publicly available on the project website. [prob]

6.4 Evaluation

We conducted an evaluation to assess the effectiveness of TMAP. In our evaluation, we address following research questions:

- **RQ1**: What is the effectiveness of TMAP in leveraging the similarity in the language of API method descriptions to discover likely API Mappings?
- **RQ2**: How do the mappings inferred by TMAP compare with the human written mappings?
- **RQ3**: What is the effectiveness of using free form queries using TMAP?

[3] `http://jsoup.org/`

6.4.1 Subjects

We evaluated TMAP using a snapshot of the API documents of Java, C#, Android, and Java ME downloaded on Jan 2015.

Java Platform Micro Edition, or Java ME, is a Java platform designed for embedded systems (such as mobile devices). Target devices range from industrial controls to mobile phones (especially feature phones) and set-top boxes. Android is a linux-based operating system designed primarily for touchscreen mobile devices, such as smartphones and tablet computers. API documents for both Java ME and Android are publicly available at `http://docs.oracle.com/javame/config/cldc/ref-impl/midp2.0/jsr118/index.html` and `http://developer.android.com/reference/packages.html` respectively.

Java and C# are general-purpose programming languages from Oracle and Microsoft respectively. Java applications are typically compiled to bytecode that run on any Java Virtual Machine (JVM) irrespective of underlying computer architecture. Likewise, C# is compiled into intermediate representation that run on Microsoft's common language infrastructure. API documents for both Java and C# are publicly available at `http://docs.oracle.com/javase/8/docs/api/` and `https://msdn.microsoft.com/en-us/library/gg145045(v=vs.110).aspx` respectively.

Particularly we used the API documents of the following library pairs as subjects for our evaluation.

Java ME (to Android) API: For our evaluation we considered the methods in the following Java ME Types as the source API methods to discover mapping methods in Android API: `Alert`, `Canvas`, `Command`, `Graphics`, and `Font` Classes in `javax.microedition.lcdui`.

The listed types provides methods for supporting graphics related functionality in Java ME. Furthermore, Gokhale et al. [GGP13] approach Rossetta reports the mapping for methods in theses types along with seven others (twelve types in total) as a part of their evaluation, thus providing for a baseline comparison with dynamic analysis based approaches. Rosetta approach requires a user to manually execute functionally similar applications using source and target API with identical (or near identical) inputs and collect execution traces. Finally Rosetta analyses the collected execution traces to infer method mappings. We focused our evaluation on the listed five types which we perceived important among the twelve types reported by Rosetta. In the future, we plan to evaluate TMAP approach on all the twelve reported types.

Java (to C#) API: For our evaluation we considered the methods in the following Java

Types as the source API methods to discover mapping methods in C# API: 1) `File`, `Reader`, and `Writer` in `java.io` package; 2) `Calendar`, `Iterator`, `HashMap`, and `ArrayList` in `java.util` package; and 3) `Connection`, `ResultSet`, and `Statement` classes in `java.sql` package.

Types in `java.io` provide the API methods for accessing and manipulating the file system. Types in `java.util` provide API methods for miscellaneous utilities, such as text manipulation, collections frameworks and other data structures. Types in `java.sql` provide the API methods for accessing and processing data stored in databases. We selected these particular packages in Java programming languages because Nguyen et al. [NNNN14] in their work (StaMiner) for statistical language migration find mappings for the types in these packages. Their mappings provides for a external baseline comparison. Although, Nguyen et al. [NNNN14] report on all the classes in these packages, due to the amount of effort, we focused our analysis on the listed types which we perceived as important in their respective packages.

6.4.2 Evaluation Setup

We first downloaded the publicly available API documents from the respective websites of the projects. We then cleaned and extracted the desired fields as described in the Section 6.3.1. We then indexed the extracted text into the Lucene indexes. We created a separate index for every API type: Java ME, Android, Java, and C#.

For every class/interface under consideration (as listed in Section 6.4.1), we extracted the publicly listed methods from API documents. For a given type, we only consider the methods that are explicitly listed in the public API. We exclude the methods that are inherited from a parent type without modification. For instance, we exclude the `equals` method from the parent class `Object` which is the parent class for majority of Java types. Since, the inherited methods are part of the parent type, TMAP will find the mapping when considering the parent type. We then use TMAP to create the queries form the descriptions of the considered methods as described in Section 6.3.2. Finally, we execute the formulated queries on the index and collect results We only consider top-10 results for each query. Previous approaches [CJS09, GGP13] also only consider the top-10 results suggested by their approach for evaluation.

The top-10 matches found by the TMAP are then analyzed/reviewed manually to determine the effectiveness of the matched results. For a given method in the source API, a match is characterized by a class and a method within that class that is determined to be the correspond-

ing implementation in the target API. Researchers [4] next annotated each match as 'relevant' and/or 'exact' based on the following acceptance criteria:

1. **Relevant** If the target method in the top-10 list can be used to implement the same (or similar) functionality as the source method, we classify the result as relevant.
   OR
   The target method is reported by the previous approaches [GGP13, NNNN14] as a mapping.

2. **Exact** If the target method is a relevant method and the target method accurately captures the functionality of the source method, and implements the same feature/function, the resultant match is classified as an 'exact' match.
   OR
   The target method is reported by the previous approaches [GGP13, NNNN14] as a mapping.

For example, the method `getInt` in the type `java.sql.ResultSet` has an exact match in the C# method `GetInt32` from `system.data. sqlclient.SqlDataReader` type, since it provides the same functionality of extracting the value stored in a specified column as a 32-bit signed integer. On the other hand, the method `getClob` in type `java.sql.ResultSet` does not have an exact corresponding method in C#. The closest functionality available is the C# method `GetValues` in the class `system.data.sqlclient.SqlDataReader`. Thus the method `GetValues` is marked as relevant, but not an exact match.

We then calculate coverage (Cov) as the ratio of the number of source methods in a type to the number of methods that TMAP found *at-least* one relevant mapping. We also calculate the Δ_{Cov} as increase in the Cov in comparison to results reported by previous approaches [GGP13, NNNN14]. High value of Δ_{Cov} indicates the effectiveness of TMAP in finding API method mappings.

Finally we measure the common methods between the exact mappings suggested by TMAP for a source method with the mappings suggested by previous approaches. We then calculate, the number of new mappings a the number of exact mappings minus the common mappings.

[4] Rahul Pandita, Raoul Jetley, Sithu Sudarsan

6.4.3 Results

We next describe our evaluation results to demonstrate the effectiveness of TMAP in leveraging natural language API descriptions to discover method mappings across APIs.

RQ1: Effectiveness of TMAP

Table 6.2 presents our evaluation results for answering RQ1. The columns 'API' lists the name of source API under 'Source' and target API under 'Target'. The column 'Type' lists the class or interface in source API under consideration for finding mappings in target API. The column 'No. Methods' lists the number of methods in the class or interface under consideration. We only consider the methods explicitly declared or overridden by a type and ignore the inherited methods. The column 'Relevant' lists the number of methods for which at least one relevant mapping is reported. The sub-column 'Prev.' reports relevancy numbers by previous approaches. The previous approach for comparison of Java ME-Android mappings is Rosetta [GGP13]. The previous approach for comparison of Java-C# mappings is StaMiner [NNNN14]. The sub-column 'TMAP' reports relevancy numbers by TMAP (at least one relevant method in top-ten results). The column 'Exact' lists the number of methods for which a exact mapping is found. The sub-column 'Prev.' reports exact numbers by previous approaches. Since previous approaches do not make a distinction between exact and relevant, we report same values for both columns. The sub-columns 'TMAP' reports exact numbers by TMAP (at least one exact method mapping in top-ten results). Column 'Δ_{Cov}' lists the ratio of increase in the number of methods for which a relevant mapping was found by TMAP to the total number of methods in the type.

Our evaluation results indicate the TMAP on an average finds relevant mappings for 57% (Column 'Δ_{Cov}') more methods. For the Java ME-Android mappings TMAP performs best for `Alert` and `Font` classes from `javax.microedition.lcdui` package in Java ME API with 75% increase in number of methods for which a relevant mapping was found in Android API. For the Java-C# mappings TMAP performs best for `Iterator` interface from `java.util` package in Java API finding a relevant method in C# API for all the methods. Previous approach StaMiner reports a manually constructed wrapper type as a mappings instead. Furthermore, our results also indicate that TMAP found on average exact mappings for 6.5 $((171 - 73)/15)$ more methods per type with a maximum of 21 additional exact mappings for a `java.sql.ResultSet` type as compared to previous approaches.

Table 6.2: Evaluation Results

S No.	API Source	API Target	Type	No. Methods	Relevant Prev	Relevant TMAP	Exact Prev	Exact TMAP	Δ_{Cov}	Common	New
1	Java ME	Android	javax.microedition.lcdui.Alert	16	3	15	3	7	0.75	0	7
2	Java ME	Android	javax.microedition.lcdui.Canvas	22	5	18	5	10	0.60	0	10
3	Java ME	Android	javax.microedition.lcdui.Command	6	3	3	3	0	0.00	0	0
4	Java ME	Android	javax.microedition.lcdui.Graphics	39	18	36	18	29	0.47	5	24
5	Java ME	Android	javax.microedition.lcdui.Font	16	3	15	3	8	0.75	0	8
6	Java	C#	java.io.File	54	15	37	15	26	0.41	7	19
7	Java	C#	java.io.Reader	10	1	8	1	6	0.70	1	5
8	Java	C#	java.io.Writer	10	2	10	2	10	0.80	1	9
9	Java	C#	java.util.Calendar*	47	0	11	0	5	0.24	0	5
10	Java	C#	java.util.Iterator*	3	0	3	0	1	1.00	0	1
11	Java	C#	java.util.HashMap	17	5	9	5	5	0.24	1	4
12	Java	C#	java.util.ArrayList	28	6	22	6	15	0.58	4	11
13	Java	C#	java.sql.Connection	52	1	28	1	13	0.52	1	12
14	Java	C#	java.sql.ResultSet	187	10	146	10	31	0.73	1	30
15	Java	C#	java.sql.Statement	42	1	21	1	5	0.48	1	4
Total				549	73	382	73	171	0.57**	22	149

*=Previous approach reported a manually constructed class as mapping; **=Average
Prev= previous approach; Previous approach for Java ME-Android mappings is Rosetta [GGP13];
Previous approach for Java-C# mappings is StaMiner [NNNN14]

We next describe the cases where TMAP did not find any relevant mapping. A major cause for inadequate performance of TMAP is lack of one-to-one mapping between methods in source and target API. Often times functionality of a method in a source API is broken down into multiple functions in the target API or vice versa. Although, TMAP reports some of the relevant methods, exact mapping may involve a sequence of method calls in target API which is the limitation of TMAP. In the future, we plan to investigate techniques to deal with such cases.

Another cause of inadequate performance of TMAP is inconsistent use of terminology across different APIs. For instance, TMAP did not find any additional relevant mapping for methods in `Command` class in Java ME API. In Java ME API, 'command' is used to refer the user interface construct 'button'. In Android API, 'command' is used in more conventional sense of the term. This inconsistent use of terminology causes TMAP to return irrelevant results. When we manually replaced the term 'command' with 'button' in the generated queries, we observed a relevant method appeared in top ten results for every method `Command` class in Java ME API. However, we refrain from including such modifications to stay true to TMAP approach for evaluation. In the future, we plan to investigate techniques to automatically suggest alternate keywords.

RQ2: Quality of discovered mappings

To answer RQ2, we compared the exact mappings discovered by TMAP with the mappings discovered by previous approaches. In Table 6.2 the previous approach for comparison of the Java ME-Android mappings is Rosetta [GGP13] and the previous approach for comparison of Java-C# mappings is StaMiner [NNNN14]. Our results show that out of 171 discovered exact mappings only 22 are in common with previous approaches. We next discuss some of the implications of the results.

Before carrying out this evaluation, we had hoped that the mappings discovered by TMAP would significantly overlap with the mappings discovered by the Rosetta and StaMiner, as these approaches infer mappings from actual source code. Thus, these mappings can be considered as the representative of how developers are actually migrating software. However, the results suggest a low overlap. We manually investigated the possible TMAP specific implications of the observed mismatch.

The results (matches found) comprise of methods from different classes in the target API, reflecting that often there are multiple ways to solve a problem, or to implement a feature using

an API. Further, choice of using multiple APIs gives a developer the flexibility to use different approaches when porting an application from one platform to another.

When more than one match is found for a given source method, the results in TMAP are ranked according to the similarity score, with the more relevant (or exact matches) ranked higher. The ranked set of results helps the developer use the best suited or most appropriate target method in their implementation. This approach is different from earlier approaches [GGP13, NNNN14] that focused on only exact matches between different classes based on the number of methods within them that are similar.

We also contacted the authors of Rosetta approach [GGP13] to report the difference in the mappings. Specifically, we inquired that Rosetta does not report any method form `AlertDialog` in the Android API as a possible mapping for `Alert.setString` method in Java ME API. Rosetta reports method sequence `Paint.setAlpha CompundButton. setChecked` as one of the likely mappings. In contrast, TMAP discovers `AlertDialog.setTitle` method in Android API as a likely mapping.

The lead author of Rosetta approach responded that they restricted the output of Rosetta to sequences of length up to 2 when inferring mappings (i.e., A− >p;q, or A− >p). Furthermore, authors count a reported method sequence as a valid mapping if the reported sequence implements some of the functionality of a source method on the target platform. With regards to our query, Rosetta authors observed that in many of the traces, setting a `string` first involves setting the attributes of the `Paint` (with is used to draw the text), followed by a call to `setText` method, which led them to believe that the sequence `Paint.setAlpha CompundButton. setChecked` could be a likely mapping, if at least in part. Although, author did confirm that technically `AlertDialog.setTitle` method in Android API as a likely mapping.

The exchange with Rosetta approach's lead author points out that that the mappings discovered by TMAP generally point to the closest aggregate API in contrast to the individual smaller API calls that achieve same functionality. Furthermore, exchange also demonstrates the reliance of the code analysis approaches on the quality of the code under analysis. In contrast, TMAP relies on the quality of the API method descriptions.

RQ3: Effectiveness in free form queries

RQ3 demonstrates the effectiveness of text-mining in general for API related information retrieval tasks. In particular, we show the effectiveness of creating vector space representation of API method descriptions by using free form queries. For evaluating RQ3, we use the same

Table 6.3: Comparison with Sniff

Query	Method Rank	
	Sniff	TMAP
get active editor window from eclipse workbench	1	1
parse a java source and create ast	1	2
connect to a database using jdbc	1	6
display directory dialog from viewer in eclipse	1	1
read a line of text from a file	1	-
return an audio clip from url	1	1
execute SQL query	2	3
current selection from eclipse workbench	1	1

'-'=No Match in top-10 results.

queries as used by Chatterjee et al.'s [CJS09] in evaluation of their approach Sniff. Sniff is targeted towards searching for relavant method snippets from source-code repositories. Sniff first annotates the source code with API document descriptions and uses the hybrid representation of source code to get relevant code fragments. Since TMAP does not report method sequences, we consider a method reported by TMAP as relevant iff:

1. the method is in top-10 results.

2. the method is one of the methods in the code fragment reported by Sniff.

Table 6.3 shows the effectiveness of TMAP using free form queries. The column 'Query' lists the query used in evaluation. The column 'Sniff' lists the ranking of relevant code fragment by Sniff as reported in their work. The column 'TMAP' lists the ranking of first relevant method returned by TMAP.

Our evaluation results show that in TMAP returns the relevant method in top 10 results (except one) demonstrating effectiveness of TMAP with free form queries. For the query 'read a line of text from a file' all the reported result methods did support the functionality of reading lines from file, however were not reported as relevant code fragment by Sniff.

6.5 Limitation and Future work

We next describe the limitations and future work of TMAP, followed by discussions on threats to validity.

A key limitation of the presented work is reliance on the human developer to confirm or refute mappings. In the future, we plan to extend the TMAP infrastructure to achieve an end-to-end automation. Particularly, we plan on using the program analysis techniques, such as type-analysis proposed in existing approaches [NNNN14, ZZXM09].

Sometimes the functionality achieved by a method call in a source API, is achieved by a sequence of method calls in the destination API and vice versa. Although TMAP may return individual methods as relevant, TMAP does not provide explicit sequences of method calls as relevant suggestions. In the future, we plan to extend the current text mining infrastructure to provide method sequences as relevant suggestions when applicable. In particular, we plan to leverage the NLP techniques, such as specification inference [PXZ+12] to identify method sequences.

From an implementation perspective, TMAP does not take into account the API fields, which limits TMAPs ability in reporting mappings involving API fields. For instance, the functionality achieved by a method call in a source API may be achieved by accessing a API field in destination API. However, disregarding API fields is a limitation of the current implementation and in future iterations of TMAP implementations we plan to include API fields in the indexes as well.

Finally, TMAP operates under the assumption of the availability of the API documents. Thus TMAP is not applicable in situations where API documents are of low quality or are unavailable altogether. In the future, we plan to extend TMAP infrastructure to workaround such situations by integrating with existing source code mining based approaches. Specifically we plan to leverage techniques like code summarisation [SPVS11] and IR based approaches like Exoa [KLHK10].

Threats to Validity: The primary threat to external validity is the representativeness of our experimental subjects to the real world software. To address this threat we chose real world API pairs: 1) Java ME and Android APIs are two Java based platforms to develop mobile applications; 2) Java and C# APIs are the top object-oriented programming language APIs used for generic application development. The threat can be further minimized by evaluating TMAP on more APIs from different domains.

Our chief threat to internal validity is the accuracy of TMAP in identifying API mappings.

To minimize this threat we compared the mappings inferred by TMAP with the mappings provided by previous approaches. We thank Gokhale et al. [GGP13] for sharing with us the Java ME and Android API mappings inferred by their Rosetta approach. We also thank Nguyen et al. [NNNN14] for making their mappings publicly available. Furthermore, we did manually identify some of the mappings that could not be compared to previous work. Thus, human errors may affect our results. To minimized the effect, each annotation was independently agreed upon by two out of three researchers (Rahul Pandita, Raoul Jetley, and Sithu Sudarsan).

6.6 Chapter Summary

API mapping across different platforms/languages mappings facilitate machine-based migration of an application from a one API to another. Thus tool assisted discovery of such mappings is highly desirable. In this chapter, we presented TMAP: a lightweight text-mining based approach to infer API mappings. TMAP compliments existing mapping inference techniques by leveraging natural language descriptions in API documents instead of relying on source code. We applied TMAP to discover API mappings for 15 types across: 1) `Java` and `C#` API, 2) `Java ME` and `Android` API. We demonstrated the effectiveness of TMAP by comparing the discovered mappings with state-of-the-art code analysis based approaches. Our results indicate that TMAP on an average found relevant mappings for 57% more methods compared to previous approaches. Furthermore, our results also indicate that TMAP found on average exact mappings for 6.5 more methods per type with a maximum of 21 additional exact mappings for a single type as compared to previous approaches.

CHAPTER 7

Related Work

Our proposed techniques encompasses multiple research areas within software engineering such as NLP on software engineering artifacts, program synthesis and software verification. We next discuss relevant work pertinent to our proposed approaches in these areas.

7.1 Formal Specifications

Design by contracts has been an influential concept in the area of software engineering in the past decade. Contracts formally specify the program behavior in terms of conditions that must hold before/after and/or during the execution of a method. A significant amount of work has been done in automated inference of code contracts. There are existing approaches that statically or dynamically extract code contracts [CTS08, NE02, TCS06]. A significant amount of work has been done in automated inference of contracts. However, studies [PCM09, FL01] demonstrate that a combination of developer-written and automatically extracted contracts is the most effective approach for formally specifying the constraints on an API.

Since developers describe the specifications in the method descriptions, we believe that our approaches can work in conjunction with existing approaches towards extracting a comprehensive set of code contracts for a method. Furthermore, Wei et al. [WFKM11] demonstrated that dynamic contract inference performed better when provided with an initial set of seed

contracts. Additionally, contracts are typically in the form of assertions on the state (member variables/ properties) of a program. In contrast, one of our proposed approaches ICON infers temporal constraints, that specify the ordering of method invocations, therefore goes beyond traditional contracts.

Another set of approaches infer code-contract-like specifications (such as behavioral model, algebraic specifications, and exception specifications) either dynamically [HRD07, GMM09, HRD08] or statically [FL01, BW08, WZ11] from source code and binaries. Among mining based approaches Gable and Su [GS08] proposed to learn *micro-patterns* of temporal properties from runtime traces and then combining them into larger specifications depicted as finite state automata. In contrast, the approaches presented in this dissertation infer specifications from the natural language text in API documents, thus complementing existing approaches when the source code or binaries of the API library is not available.

7.2 NLP in Software Engineering

NLP techniques are increasingly applied in the software engineering domain. NLP techniques have been shown to be useful in requirements engineering [SPKB09, SSP10, GZ05, KPW15], usability of API documents [DH09], and other areas [ZCY08, LM07]. We next describe most relevant approaches.

- *Access control policies*: Xiao et al. [XPTX12] and Slankas et al. [SW13] 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, due 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. [ZZXM09] 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, approaches presented in this

dissertation 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. [TYKZ07] 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. [TMTL12b] extended the work of Tan et al. [TYKZ07] 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-like representation of the specifications that can be used to test implementation.

- *Requirements Document*: Sinha et al. [SPKB09, SSP10] 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.

7.3 Program Comprehension

With respect to program comprehension there are existing techniques that assist in building domain specific ontologies [ZCY08]. Furthermore, there are existing approaches [SPVS11, Rob86] that automatically infer natural language documentation from source code. These approaches would immensely help in comprehension of the functionality of an application. How-

ever 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. [SPVS11] 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. [ZZE11] 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` is translated to *x is set to:null*.

- Buse et al. [BW08] 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.

7.4 Program Synthesis

Automated program synthesis has been gaining traction with a lot of recent work [Gul11, GJTV11, HG11, GKT11, IGIS10, JGST10, SGF10b, TGT11, TSSC12] in this direction. Srivastava et. al. [SGF10b] addresses the problem by leveraging specifications in the form of pre/post conditions and invariant to achieve synthesis. There also exists some work [HG11, IGIS10, TGT11, JGST10] 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. [TSSC12] 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.

7.5 Language Migration

Language migration is an active area of research [HH05, Mos03, vDK99, Wat88, ZTX+10, GGP13, NNNN14], with myriad techniques that have been proposed over time to achieve automation. However, most of these approaches focus on syntactical and structural differences across languages. For instance, Deursen et al. [vDK99] proposed an approach to automatically infer objects in legacy code to effectively deal with differences between object-oriented and procedural languages.

However, El-Ramly et al. [EREA06] points out that most of these approaches support only a subset of API's for migration. Another recently published survey by Robillard et al. [RBK+13] provides a detailed overview of techniques dealing with mining API mappings. Among other works described in [RBK+13], Mining API Mapping [ZTX+10] (MAM) is most directly related to our work.

MAM mines API mapping relations across different languages for language migration, however there is a significant difference between MAM and TMAP. MAM takes into input a software (S) written in source language and manually ported version of S (S') written in target language. MAM then applies a technique called "method alignment" that pairs the methods with similar functionality across S and S'. These methods are then statically analyzed to detect mappings between source and target language API. While MAM requires as input software that has been manually ported from a source to target API (both S and S'), our approach is independent of such requirement. In contrast, TMAP relies on text mining of source and target API method descriptions (that are typically publicly available) to discover likely mappings.

Gokhle et al.'s [GGP13] approach Rosetta addresses the limitations of MAM to infer method

mappings. In particular, Rosetta relaxes the constraint of having software that has been manually ported from a source to target API. Instead, they use functionally similar software in source and target API. For instance, they use two different 'tic-tac-toe' game applications in Java ME and Android API not necessarily manually ported. Rosetta approach then requires user to manually execute these applications under identical (or near identical) inputs and collect execution traces. Finally Rosetta analyses the collected execution traces to infer method mappings. In contrast, TMAP is independent of both the requirements: 1) to have functionally similar applications, 2) to manually execute such applications using similar inputs.

Recently Nguyen et al. [NNN14, NNNN14] proposed StaMiner, an approach that applies statistical machine translation based techniques to achieve language migration. Their approach builds upon a previous result [HBS+12] that demonstrates the effectiveness of using an n-gram model to predict the next token in software source file given a large corpus of software source files to learn from. Similar to MAM [ZTX+10] approach they require a software S written in source language and manually ported version of S' written in target language as input. They then consider source code as a sequence of lexical tokens (lexemes) and apply a phrase-based statistical machine translation model on the lexemes of those tokens. In contrast, TMAP is independent of such requirement (presence of S and S').

Furthermore, from an infrastructure perspective, TMAP is independent of the programming language or API under consideration. In contrast, program analysis based approaches like [ZTX+10, GGP13, NNNN14] may need significant efforts for adding support to additional APIs and programming language.

Zheng at al. [ZZL11] mine search results of a web search engine, such as Google to recommend related APIs of different libraries. In particular, they propose heuristics to formulate keywords using the name of the source API, and the name of target API to query a web search engine. For instance, to search for an equivalent class in C# for the `HashMap` class in Java, a user may manually enter "HashMap C#" in a web search engine. The results are computed one by one and candidates are ranked by relevance, mainly according to their frequency of the appearance of keywords in the query. However, authors provides only preliminary results and queries proposed are of a coarse grain. Furthermore, the results are susceptible to influence by the webpages returned by a web search engine. In contrast, TMAP is independent of the web search engine results.

7.6 Information Retrieval

Information retrieval techniques [CJS09, GFX+10, KLHK10, Rei09] are also being increasingly used in Code Search. We next describe some relevant approaches. Chatterjee et al.'s [CJS09] approach Sniff annotates the source code with API document descriptions. Sniff then performs additional type analysis on the source code to rank relevant code snippets. Grechanik et al.'s [GFX+10] approach Exemplar uses the text in API documents to construct a set of keywords associated with an API call. Exemplar then uses the keywords list to facilitate query expansion to achieve code search. However, these approaches are targeted towards code search in a one API. In contrast, TMAP discovers method mappings across multiple APIs.

7.7 Query Formulation

Shepherd et al. [SFH+07] developed tool V-Do to extract verb direct objects from source code comments and identifiers. The extracted verb-direct objects are provided as recommendations to the developer performing the queries. Haudic et al. [HBM+13] developed techniques to automatically recommend query alternatives by learning from past-queries, using ML based reproaches on historical query data. Hill et al. [HPVS09] proposed a query expansion mechanism by automatically extracting natural language phrases from the source code to be searched.

CHAPTER 8

Future Work

This dissertation presented natural language processing and text mining based approaches that are targeted towards developer, tester and end user productivity. However, there are also many opportunities for future work. This chapter outlines some of the future work.

8.1 Generic Limitations

Several limitations exists in the presented approaches, that have been individually discussed for each approach. These limitations provide an opportunity to expand the presented work in future. From an practitioners perspective, following areas of our approach need further work.

- **Elimination of Predefined Lists**. The presented approaches uses predefined lists for domain dictionaries. However, there are approaches [ZCY08] that facilitate building domain dictionaries from source code. We plan to extend our work to use these approaches. Furthermore, we rely on pre-defined templates for code contract generation. While such a strategy serves our purpose of prototyping, advanced techniques such as keyword programming [LM07] have shown promising results in building programming statements using keywords. We plan to explore such techniques and evaluate the overall effectiveness of our approach after augmenting it with such techniques.

- **Extending Generic Dictionaries**. Our approaches often uses generic dictionaries like Wordnet for synonym analysis (for calculating semantic equivalence). The use of generic dictionaries for software engineering related text is sometimes inadequate. For instance, Wordnet matches "has" as a synonym for the word "get". Although valid for generic English, such instances cause our approach to incorrectly distinguish a constraint sentence from a regular sentence, or vice versa. In future work, we plan to investigate techniques to extend generic dictionaries for software engineering related text. In particular, Yang and Tan [YT13] recently proposed a technique for inferring semantically similar words from software context to facilitate code search. We plan to explore such techniques and evaluate the overall effectiveness of our approach after augmenting it with such techniques.

- **Multi-sentence specifications**. Our approach currently takes into account the specifications described in a single sentence. However, there are instances when a specification is distributed across several sentences. Consider the case for parameter specification described in sentences below:

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

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

- **Results Presentation**: A typical user of this system may be interested in finding the source of the inferred specifications in the software artifacts. With advancement of techniques presented in this work, a potential side-effect might be more verbose software artifacts describing detail specifications in natural language. One can argue that it would lead to additional burden on users to review a lengthy specification sentences. However, such additional description provides an opportunity for the users to make informed decisions instead of making assumptions. In future work, we plan to implement and evaluate interfaces to present users with information in a more efficient way, countering user

fatigue in case of lengthy descriptions. For example, we may consider using icons in different colors to represent different type of specification sentences, building on the work of Dekel and Herbsleb [DH09] who were the first to create a tool namely eMoose, an Eclipse IDE based plug-in that allowed developers to create directives (way of marking the specification sentences) in the default API documentation. These directives are highlighted whenever they are displayed in the Eclipse environment.

8.2 Coding-specific activities

The presented approaches serve as a way to formalize the description of specifications in the natural language texts of API documents, thus facilitating existing tools to process these specifications. We next discuss the benefits of our approach in other areas of software engineering,

- **Code Searching**. Code searching [TX07, Rei09] for reuse is a classic problem [FK05] in software engineering. Among previous approaches, a recent approach by Riess [Rei09] provides promising results by using semantics such as code contracts as input-output relationships for code searching. Our approach can be used for generating specifications from API documents in a code repository and thus assisting such approaches in producing better results.

  In case of mobile applications, our WHYPER framework currently takes into account application descriptions and Android API documents to highlight permission sentences. Thus, our framework can semi-formally enumerate the uses of a permission. This information can be leveraged in the future to enhance searching for desired applications.

  Furthermore, the outputs from our framework could be used in conjunction with program analysis techniques to facilitate effective code reuse. For instance, our framework outputs the reasons of why a permission is needed for an application. These reasons are usually the functionalities provided by the application. In future work, we plan to locate the code fragments that implement the described functionalities. Such mapping can be used as indexes to existing code searching approaches [TX07, Rei09] to facilitate effective reuse.

- **Program Synthesis**. Automated program synthesis holds potential for easing the task of a developer by taking care of program generation and allowing the developer to concentrate on design tasks. Recent work by Srivastava et al. [SGF10a] addresses the problem

by leveraging specifications in the form of pre/post-conditions and invariants to achieve synthesis. Our approach can work in conjunction with such approaches to extract specifications from natural language text to achieve better synthesis.

8.3 Self Assurance

A key assumption made by the presented approaches is that the software artifacts faithfully and accurately describe correct specifications. Inaccuracies in software artifacts may negatively affect the usability of the presented approaches. Thus, the presented approaches should be extended to perform some form of self assurance to assess the validity of inferred specifications.

For instance, API documents can sometimes be misleading [TMTL12a, RGL10], thus causing developers to write faulty client code. In future work, we plan to extend our approach to find documentation-implementation inconsistencies. Likewise, A key limitation of the TMAP work is reliance on the human developer to confirm or refute mappings. In the future, we plan to extend the TMAP infrastructure to achieve an end-to-end automation. Particularly, we plan on using the program analysis techniques, such as type-analysis proposed in existing approaches [NNNN14, ZZXM09].

8.4 WHYPER : Generalization to Other Permissions

WHYPER is designed to identify the textual justification for permissions that protect "user understandable" information and resources. That is, the permission must protect an information source or resource that is in the domain of knowledge of general smartphone users, as opposed to a low-level API only known to developers. The permissions studied in Chapter 5 (i.e., address book, calendar, microphone) fall within this domain. Based on our studies, we expect similar permissions, such as those that protect SMS interfaces and data stores, the ability to make and receive phone calls, read call logs and browser history, operate and administer Bluetooth and NFC, and access and use phone accounts will have similar success with WHYPER.

Due to current developer trends and practices, there is class of permissions that we expect will raise alarms for many applications when evaluated with WHYPER. Recent work [EGC+10, EOMC11] has shown that many applications leak geographic location and phone identifiers without the users knowledge. We recommend that deployments of WHYPER first focus on other permissions to better gauge and account for developer response. Once general deploy-

ment experience with WHYPER has been gained, these more contentious permissions should be tackled. We believe that adding justification for access to geographic location and phone identifiers in the application's textual description will benefit users. For example, if an application uses location for ads or analytics, the developer should state this in the application description.

Finally, there are some permissions that are implicitly used by applications and therefore will have poor results with WHYPER. In particular, we do not expect the Internet permission to work well with WHYPER. Nearly all smartphone applications access the Internet, and we expect attempts to build a semantic graph for the Internet permission will be largely ineffective.

8.5 Long term goal

My research, in general, is targeted to develop tools and techniques for improving software quality. In the future, I would also like to work on the area of *automated specifications inference* investigating competing approaches such as static and dynamic program analysis. Finally, a long-term goal of mine is to develop tools and techniques to achieve *assisted program synthesis*. I envision *assisted program synthesis* as a way of writing programs, where a human is focused on the semantic and creative aspect of the problem solving and a computer-system assists human by taking care of low level details such as programming language syntax and configuration issues.

CHAPTER 9

Summary and Lessons Learned

In this chapter, we summarize the contributions of this dissertation and discuss the important lessons learned during the course of this dissertation.

9.1 Summary

Specifications play an important role in ensuring timely-delivered and high quality software. Natural language software artifacts are valuable source of finding specifications. For instance, documentation of API methods contains detailed specifications of how to correctly use methods. However, specifications described in natural language are often not amenable to existing developer productivity tools (that currently operate on formal specifications). This dissertation presented a number of text-analysis based approaches to infer formal specifications from natural language software artifacts.

We presented approaches to infer method specifications (parameter specifications, return value specifications and temporal specifications) from API documents. Chapter 3 presented a novel approach for inferring formal parameter specifications and return value specifications from API documents targeted towards code contract generation. Our evaluation results show that our approach has an average of 92% precision and 93% recall in identifying sentences describing parameter specifications and return value specifications from 2717 sentences. Fur-

thermore, our results also show that our approach has an average of 83.4% accuracy in inferring formal specifications from sentences describing parameter specifications and return value specifications out of 1691 sentences.

Chapter 4 presented ICON an approach to infer temporal constraints from natural language text of API documents. We used ICON to infer temporal constraints from the `PayPal Payment REST` API, the `Amazon S3 REST` API, and the commonly used package `java.io` in the JDK API. Our evaluation results show that ICON effectively identifies sentences describing temporal constraints with an average 79% precision and 60% recall, from more than 4000 sentences in subject API documents. Furthermore, ICON also achieves an accuracy of 70% in inferring 77 formal temporal constraints from these temporal constraint sentences.

We also presented an approach to infer method mapping across API's by analyzing natural language method descriptions across APIs. API mapping across different platforms/languages mappings facilitate machine-based migration of an application from a one API to another. In Chapter 6, we presented TMAP: a lightweight text-mining based approach to infer API mappings. TMAP compliments existing mapping inference techniques by leveraging natural language descriptions in API documents instead of relying on source code. We applied TMAP to discover API mappings for 15 types across: 1) `Java` and `C#` API, 2) `Java ME` and `Android` API. We demonstrated the effectiveness of TMAP by comparing the discovered mappings with state-of-the-art code analysis based approaches. Our results indicate that TMAP on an average found relevant mappings for 57% more methods compared to previous approaches. Furthermore, our results also indicate that TMAP found on average exact mappings for 6.5 more methods per type with a maximum of 21 additional exact mappings for a single type as compared to previous approaches.

Finally, in Chapter 5, we presented WHYPER: an approach to automate the task of locating the need for security permission in a mobile application description. WHYPER uses Natural Language Processing (NLP) techniques to determine why an application uses a permission. We evaluated our prototype implementation of WHYPER on real-world application descriptions that involve three permissions (address book, calendar, and record audio). These are frequently-used permissions that protect privacy and security sensitive resources. Our evaluation results show that WHYPER achieves an average precision of 82.8%, and an average recall of 81.5% for three permissions. In summary, our results demonstrate great promise in using NLP techniques to bridge the semantic gap of user expectations to aid the risk assessment of mobile applications.

In summary, our evaluation results demonstrate great promise in using text analysis techniques in aiding with software engineering tasks.

9.2 Lessons learned

This section describes the lessons learned during the course of the research performed towards this dissertation. We hope these would be useful to other researchers, including us, in their future research.

- **Personal pain-points**: The motivation behind performing this work came form the personal experiences. As a new developer, I seldom read API documents and sometimes made incorrect assumptions about using an API method. Such issues were hard to resolve because API documents were unyielding in terms of searching for that one sentence in a verbose description. Driving research from personal pain-point allowed me stay motivated throughout the process. Furthermore, talking to other people about the pain-point facilitated a deeper understanding and a problem-driven approach towards research.

- **Adapting techniques from other research areas**: When existing techniques from a different research area are applied to software engineering problems, they often face difficulties. These difficulties provide an opportunity to perform novel-research and creative-thinking. For instance, existing natural language processing techniques are developed for well written news articles. Thus, these techniques had to be creatively adapted for software engineering artifacts like API documents that are domain specific.

- **Generalize research ideas**: Oftentimes working on research problems, I became very focused on solving a very specific problem. In retrospect, the proposed techniques may very well be generalized. For instance, this research began with the idea of inferring code-contracts for C# programming language : a very specific program analysis framework for a specific programming language. However, the general idea is programming language and analysis framework agnostic, as presented in this dissertation.

- **Building on top of existing industrial tools**: Building prototypes for evaluation of a research idea is always desirable. Since writing code for everything from scratch may not be very practical, I highly recommend using existing tools and libraries whenever applicable. I also recommend building on top of widely used tool to have greater impact.

For instance, as a PhD student, I developed a prototype tool [PXTdH10] to direct Pex [1], a dynamic-symbolic-execution based automated-test-input-generation tool, to generate test inputs to achieve high logic coverage. Pex was built to target path coverage thus the inputs generated by Pex performed poorly in achieving high logic coverage. Since Pex is a popular test input generation framework and people were seeking ways to achieve logic coverage, our research inspired a test input generation strategy within Pex to target logic coverage [TdHX14].

---

[1] `http://research.microsoft.com/en-us/projects/pex/`

REFERENCES

[ama] Amazon S3 - Two Trillion Objects, 1.1 Million Requests / Second. `http://aws.typepad.com/aws/2013/04/amazon-s3-two-trillion-objects-11-million-requests-second.html`.

[AZHL12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proc. 19th CCS*, pages 217–228, 2012.

[BKvOS10] David Barrera, H Güneş Kayacik, Paul C van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proc. of the 17th CCS*, pages 73–84, 2010.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Proc. CASSIS, LNCS vol. 3362*, 2004.

[Bog00] Branimir K. Boguraev. Towards finite-state analysis of lexical cohesion. In *Proc. FSMNLP*, 2000.

[BW08] Raymond P.L. Buse and Westley R. Weimer. Automatic documentation inference for exceptions. In *Proc. 17th ISSTA*, pages 273–282, 2008.

[BW12] Raymond PL Buse and Westley Weimer. Synthesizing API usage examples. In *Proc. 34th ICSE*, pages 782–792, 2012.

[Car96] Jean Carletta. Assessing agreement on classification tasks: the kappa statistic. *Computational linguistics*, 22(2):249–254, 1996.

[Cha06] P. Chalin. Are practitioners writing contracts? *The RODIN Book LNCS*, 4157(7):100, 2006.

[CJS09] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009.

[CRTE13] Saurabh Chakradeo, Brad Reaves, Patrick Traynor, and William Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *Proc. 6th WiSec*, 2013.

[CTS08] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proc. 30th ICSE*, pages 281–290, 2008.

[DH09] Uri Dekel and James D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.

[dMM08] Marie Catherine de Marneffe and Christopher D. Manning. The stanford typed dependencies representation. In *Workshop COLING*, 2008.

[dMMM06] Marie Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *Proc. LREC*, 2006.

[DRS+09] 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.

[ea98] Fellbaum et al. *WordNet: an electronic lexical database.* Cambridge, Mass: MIT Press, 1998.

[EGC+10] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of 9th USENIX OSDI*, pages 1–6, 2010.

[EKKV11] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proc. of 18th ISOC NDSS*, 2011.

[EOM09] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proc. of 16th ACM CCS*, 2009.

[EOMC11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *Proc. 20th USENIX Security Symposium*, page 21, 2011.

[EREA06] M. El-Ramly, R. Eltayeb, and H.A. Alla. An experiment in automatic conversion of legacy Java programs to C#. In *Proc. IEEE CSA*, pages 1037–1045, 2006.

[FCH+11] A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of 18th ACM CCS*, pages 627–638, 2011.

[FFC+11] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *Proc. ACM Workshop on SPSM)*, 2011.

[FGM05] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proc, 43nd ACL*, 2005.

[FGW11] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proc. of the 2nd USENIX conference on Web application development*, pages 7–7, 2011.

[FHE+12] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension and Behavior. In *Proc. of SOUPS*, 2012.

[FK05] W.B. Frakes and Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529 – 536, 2005.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proc. 10th FME*, pages 500–517, 2001.

[Fra92] WB Frakes. Introduction to information storage and retrieval systems. *Space*, 14:10, 1992.

[GC92] J.E. Gaffney and R.D. Cruickshank. A general economics model of software reuse. In *Proc. 14th ICSE*, pages 327 – 337, 1992.

[GCEC12] Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proc. of 5th TRUST*, pages 291–307, 2012.

[GFX+10] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 475–484. IEEE, 2010.

[GGP13] Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. Inferring likely mappings between APIs. In *Proc. 35nd ICSE*, 2013.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46-6:62–73, 2011.

[GKT11] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *ACM SIGPLAN Notices*, volume 46-6, pages 50–61, 2011.

[GMM09] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *Proc. 31st ICSE*, pages 430–440, 2009.

[Gre99] Grefenstette Gregory. *Light Parsing as Finite State Filtering.* Cambridge University Press, 1999.

[GS08] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proc. 16th FSE*, pages 339–349, 2008.

[Gul11] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46-1, pages 317–330, 2011.

[GZ05] Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transactions Software Engineering Methodologies*, 14:277–330, 2005.

[GZZ+12] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and accurate zero-day Android malware detection. In *Proc. of 10th MobiSys*, pages 281–294, 2012.

[HBM+13] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 842–851. IEEE Press, 2013.

[HBS+12] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[HFB+08] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori 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.

[HG11] William R Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, volume 46-6, pages 317–328, 2011.

[HH05] Ahmed E. Hassan and Richard C. Holt. A lightweight approach for migrating web frameworks. *Inf. Softw. Technol.*, 47(8):521–532, Jun 2005.

[HHJ+11] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David 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.

[HK06] Jiawei Han and Micheline Kamber. *Data Mining, Southeast Asia Edition: Concepts and Techniques*. Morgan kaufmann, 2006.

[HPVS09] Emily Hill, Lori Pollock, and K Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings*

of the 31st International Conference on Software Engineering, pages 232–242. IEEE Computer Society, 2009.

[HRD07] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Discovering documentation for java container classes. *IEEE Trans. on Software Engineering*, 33:526–543, 2007.

[HRD08] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Developing and debugging algebraic specifications for java classes. *ACM Trans. Softw. Eng. Methodol.*, 17(3):14:1–14:37, 2008.

[HYG+13] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, and Robert Deng. Comparing mobile privacy protection through cross-platform applications. In *Proc. of the 20th NDSS*, 2013.

[IGIS10] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. In *ACM Sigplan Notices*, volume 45-10, pages 36–46, 2010.

[jav] Java 2 CSharp Translator for Eclipse. `http://sourceforge.net/projects/j2cstranslator/`.

[JGST10] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proc. 32nd ICSE*, volume 1, pages 215–224, 2010.

[kas] Kaspersky Labs- Android Malware report 2012 . `http://www.kaspersky.com/about/news/press/2012/Android_Under_Attack__Malware_Levels_for_Googles_OS_Rise_Threefold_in_Q2_2012`.

[KLHK10] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Towards an intelligent code search engine. In *AAAI*, 2010.

[KM03a] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proc. 41st ACL*, pages 423–430, 2003.

[KM03b] Dan Klein and D. Manning, Christopher. Accurate unlexicalized parsing. In *Proc. 41st Meeting of the Association for Computational Linguistics*, pages 423 – 430, 2003.

[KM03c] Dan Klein and D. Manning, Christopher. Fast exact inference with a factored model for natural language parsing. In *Proc. 15th NIPS*, pages 3 – 10, 2003.

[KPW15] Jason King, Rahul Pandita, and Laurie Williams. Enabling forensics by proposing heuristics to identify mandatory log events. In *Proc. of the 2015 Symposium and Bootcamp on the Science of Security*, page 6. ACM, 2015.

[LAH+12] Jialiu Lin, Shahriyar Amini, Jason I Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proc. of the 14th ACM UbiComp*, pages 501–510, 2012.

[LJMR12] Choonghwan Lee, Dongyun Jin, PO Meredith, and Grigore Rosu. Towards categorizing and formalizing the JDK API. *Technical Report http://hdl.handle.net/2142/30006, Department of Computer Science, University of Illinois at Urbana-Champaign*, 2012.

[LM07] Greg Little and Robert C. Miller. Keyword programming in Java. In *Proc. 22nd ASE*, pages 84–93, 2007.

[Loc12] Hiroshi Lockheimer. Android and Security. Google Mobile Blog, February 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[LPC+11] Heeyoung Lee, Yves Peirsman, Angel Chang, Nathanael Chambers, Mihai Surdeanu, and Dan Jurafsky. Stanford's multi-pass sieve coreference resolution system. In *Proc. CoNLL-2011 Shared Task*, 2011.

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

[mal] McCallum, Andrew Kachites. "MALLET: A Machine Learning for Language Toolkit.". http://mallet.cs.umass.edu.2002.

[ME10] Patrick McDaniel and William Enck. Not So Great Expectations: Why Application Markets Haven't Failed Security. *IEEE Security & Privacy Magazine*, 8(5):76–78, September/October 2010.

[Mey92] B. Meyer. Applying 'design by contract'. *IEEE Transactions on Computer*, 25(10):40 –51, oct 1992.

[Mos03] Maxim Mossienko. Automated Cobol to Java recycling. In *Proc. 7th CSMR*, pages 40–, 2003.

[MRS08] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proc. ISSTA*, pages 232–242, 2002.

[NNN14] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 544–547. ACM, 2014.

[NNNN14] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 457–468. ACM, 2014.

[NW06] David G. Novick and Karen Ward. Why don't people read the manual? In *Proc. 24th ACM*, pages 11–18, 2006.

[Ols08] D. Olson. *Advanced data mining techniques*. Springer Verlag, 2008.

[PCM09] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proc. 18th ISSTA*, pages 93–104, 2009.

[PGS+12] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proc. 19th CCS*, 2012.

[proa] ICON. https://sites.google.com/site/temporalspec.

[prob] TMAP. https://sites.google.com/a/ncsu.edu/apisim/.

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

[PXTdH10] Rahul Pandita, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Guided test generation for coverage criteria. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[PXY+13] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: towards automating risk assessment of mobile applications. In *Proc. 22nd USENIX conference on Security*, pages 527–542, 2013.

[PXZ+12] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. 34th ICSE*, 2012.

[R+11] Philip Russom et al. Big data analytics. *TDWI Best Practices Report, Fourth Quarter*, 2011.

[RB10] B. Rubinger and T. Bultan. Contracting the Facebook API. In *4th AV-WEB*, pages 61–72, 2010.

[RBK+13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Trans. on Software Engineering*, 39(5):613–637, 2013.

[Rei09] Steven P. Reiss. Semantics-based code search. In *Proc. 31st ICSE*, pages 243–253, 2009.

[RGL10] Cindy Rubino-González and Ben Liblit. Expect the unexpected: Error code mismatches between documentation and the real world. In *Proc. 9th PASTE*, pages 73–80, 2010.

[RLR+10] Karthik Raghunathan, Heeyoung Lee, Sudarshan Rangarajan, Nathanael Chambers, Mihai Surdeanu, Dan Jurafsky, and Christopher. D Manning. A multi-pass sieve for coreference resolution. In *Proc. EMNLP*, 2010.

[Rob86] P.N. Robillard. Schematic pseudocode for program constructs and its computer automation by schemacode. *Comm. of the ACM*, 29(11):1072–1089, 1986.

[SFH+07] David Shepherd, Zachary P Fry, Emily Hill, Lori Pollock, and K Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224. ACM, 2007.

[SGF10a] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proc. 37th POPL*, pages 313–326, 2010.

[SGF10b] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. From program verification to program synthesis. In *ACM Sigplan Notices*, volume 45-1, pages 313–326, 2010.

[Sin01] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[SNL99] The Stanford Natural Language Processing Group, 1999. `http://nlp.stanford.edu/`.

[SPKB09] Avik Sinha, Amit M. Paradkar, Palani Kumanan, and Branimir 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.

[SPVS11] 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.

[SSP10] Avik Sinha, Stanley M. SuttonJr., and Amit Paradkar. Text2test: Automated inspection of natural language use cases. In *Proc. ICST*, pages 155–164, 2010.

[ST97] Mark Stickel and Mabry Tyson. *FASTUS: A Cascaded Finite-state Transducer for Extracting Information from Natural-language Text.* MIT Press, 1997.

[SW13] John Slankas and Laurie Williams. Access control policy extraction from unconstrained natural language text. In *Proc. PASSAT*, 2013.

[TCS06] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering likely method specifications. In *Proc. 8th ICFEM*, pages 717–736, 2006.

[TdHX14] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an automated test generation tool to practice: From pex to fakes and code digger. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 385–396. ACM, 2014.

[TGT11] Ankur Taly, Sumit Gulwani, and Ashish Tiwari. Synthesizing switching logic using constraint solving. *International journal on software tools for technology transfer*, 13-6:519–535, 2011.

[TMTL12a] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing javadoc comments to detect comment-code inconsistencies. In *Proc. 5th ICST*, April 2012.

[TMTL12b] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing javadoc comments to detect comment-code inconsistencies. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, April 2012.

[TSSC12] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. Automating test automation. In *Proc. 34th ICSE*, pages 881–891, 2012.

[TX07] Suresh Thummalapenta and Tao Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. 22nd ASE*, pages 204–213, 2007.

[TYKZ07] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In *21st SOSP*, pages 145–158, 2007.

[vDK99] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proc. 21st ICSE*, pages 246–255, 1999.

[Wat88] Richard C Waters. Program translation via abstraction and reimplementation. *IEEE Trans. on Software Engineering*, 14(8):1207–1228, 1988.

[WDZ+13] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proc. 10th Working Conference on MSR*, pages 319–328, 2013.

[WFKM11] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proc. 33rd ICSE*, pages 474–484, 2011.

[WPF+10] 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.

[WWL+13] Qian Wu, Ling Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Inferring dependency constraints on parameters for web services. In *Proc. 22nd WWW*, pages 1421–1432, 2013.

[WZ11] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

[XPTX12] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proc. 20th FSE*, 2012.

[YT13] Jinqiu Yang and Lin Tan. SWordNet: Inferring semantically related words from software context. *Empirical Software Engineering*, 2013.

[YXP+14] Wei Yang, Xusheng Xiao, Rahul Pandita, William Enck, and Tao Xie. Improving mobile application security via bridging user expectations and application behaviors. In *Proc. of Symposium and Bootcamp on the Science of Security*, page 32. ACM, 2014.

[YY12] Lok Kwong Yan and Heng 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.

[ZCY08] Hong Zhou, Feng Chen, and Hongji Yang. Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology. In *Proc. 8th QSIC*, pages 225 –234, 2008.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. of IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

[ZTX+10] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining API mapping for language migration. In *Proc. 32nd ICSE*, pages 195–204, 2010.

[ZWZJ12] 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.

[ZXZ+09] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending API usage patterns. In *Pro. 23rd ECOOP*, pages 318–343, 2009.

[ZZE11] Sai Zhang, Cheng Zhang, and Michael D Ernst. Automated documentation inference to explain failed tests. In *Proc. 26th ASE*, pages 63–72, 2011.

[ZZL11] Wujie Zheng, Qirun Zhang, and Michael Lyu. Cross-library API recommendation using web search engines. In *Proc. 13th ESEC/FSE*, pages 480–483, 2011.

[ZZXM09] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Proc. 24th ASE*, pages 307–318, 2009.