Automatically Inferring Specifications from API Documents to Detect Bugs

Author
Institution
First line of institution address
Second line of institution address
SecondAuthor@institution.com

Abstract

The ABSTRACT is to be in fully-justified italicized text, at the top of the left-hand column, below the author and affiliation information. Use the word "Abstract" as the title, in 12-point Times, boldface type, centered relative to the column, initially capitalized. The abstract is to be in 10-point, single-spaced type. The abstract may be up to 3 inches (7.62 cm) long. Leave two blank lines after the Abstract, then begin the main text.

1 Introduction

API documents are often provided with libraries either through online accesses or through printed copies. For example, the J2SE library provides its API documents in structured HTML files generated by Javadoc[1] through its website[2]. Existing surveys [6,9] show that these documents are quite useful to programmers. However, due to various factors, documents can also mislead programmers to produce bugs (see Section 2 for details). In particular, there are inconsistencies between documents and libraries' implementations, which indicates inaccurate documents or bugs in libraries. There are also inconsistences between documents and programmers' understanding of libraries, which may introduce bugs in client code.

As the inconsistencies can introduce bugs, it is desirable to find these inconsistencies automatically for bugs. One barrier to find these inconsistences is that API documents are often written in natural languages. Due the limits of natural language processing (NLP) [17], it is virtually impossible to infer rules from arbitrary documents, but it is feasible to infer rules from documents that satisfy some predefined templates. In particular, some documents explicitly specify the usages of libraries. For example, one document for `javax.naming.ldap.StartTlsResponse` says that "*setHostnameVerifier() must be called before negotiate() is invoked for it to have effect* ". iComment developed by Tan et al. [23] first finds out documents that explicitly contain such rules and uses templates to infer rules from these rule-containing documents. In this example, the corresponding template is that "$\langle F_A \rangle$ must (NOT) be called before $\langle F_B \rangle$". From the preceding document and its corresponding template, the specification `setHostnameVerifier()` $\rightarrow$ `negotiate()` can be inferred to detect bugs.

Using specifications inferred from documents, Tan et al. [23] have found some real bugs for projects such as Linux, Mozilla, Wine, and Apache. However, many documents do not explicitly contain any rules, and these documents simply describe what kind of *action* the method takes against a particular *resource*. For example, the document for `javax.sound.midi.MidiDevice.open()` is that "*opens the device, indicating that ...*", whereas the document for `javax.sound.midi.MidiDevice.close()` is that "*closes the device, indicating that ...*". Although the two documents do not explicitly contain rules, it is feasible to infer some implicit rules from their actions (open and close) against the same resource (device). In this paper, we present an approach and its support tools to infer specifications from documents that do not explicitly contain any rules. The main contribution of our work is as follows.

- We propose an approach to infer specifications from documents that do not explicitly contain any rules.
- The approach calls for the attention to detect the bugs that are introduced by the inconsistencies among libraries' implementations, API documents, and programmers' understanding automatically.
- We develop XX and conduct an experiment on XX, XX, and XX. Totally, we find XX bugs in libraries, XX bugs in API documents, and XX bugs in clients.

In remainder of this paper is organized as follows. Section 2 introduces real bugs that are related to API documents. Section 3 introduces the resource model to infer

[1] http://java.sun.com/j2se/javadoc/
[2] http://java.sun.com/j2se/1.5.0/docs/api/

specifications. Section 4 presents the implementation of XXX. Section 5 presents our experiment. Section 6 discusses the issues in our approach. Section 7 introduces the related work, and Section 8 concludes.

2 The Related Bugs

This section shows some confirmed real bugs that are related to API documents. These bugs are from the J2SE bug database[3] and the standard bug patterns used by FindBugs[4].

Inaccurate documents. This kind of bugs are introduced when the documents fail to describe the libraries' implementation accurately. For example, Bug #4118434 says that the behavior of `java.net.Socket.close()` is inconsistent with its document, and the difference needs to be specified. Bug #2026017 says that the description of `ROI.performImageOp()` is inaccurate. The situation is worsened by software evolution and growth when documents become out-of-date. For example, Bug #6654823 says that the description of `JnlpDownloadServlet` is out-of-date with errors. Bug #6444133 says that the documents of `javax.swing.JList` is out-of-date and needs to be updated.

Bugs in libraries' implementations. This kind of bugs are introduced when libraries' implementations fail to meet their documents in some certain situations. For example, Bug #4171239 says that `File.deleteOnExit()` fails to delete a file when the file is open. Bug #4673298 says that `java. nio.channels.FileChannel.lock()` fails to lock files on NFS mounted drivers.

Bugs in client code. When programmers use inaccurate API documents to develop client code, bugs may be introduced as the programmers may have a wrong understanding on how to use libraries. When there are bugs in libraries' implementations, these bugs can cause bugs in client code as libraries do not work correctly. Even if documents are accurate and libraries' implementations contain no bugs, careless programmers still violate these correct usages and introduce bugs. For example, many API documents of database drivers explicitly warn the disaster consequence of not closing opened connections. In particular, Oracle*9i* JDBC documents [20] have the warning of "*If you do not explicitly close your ResultSet and Statement object, serious memory leaks could occur*". Still, many programmers forget to close these objects and produce bugs. These bugs are known as the ODR bugs in FindBugs, whose description is that "*ODR: Method may fail to close database resource*". Existing researches on programmers's behaviors [18, 19] show that programmers are reluctant to read manuals such as API documents. This behavior may explains why programmers still produce bugs even if documents are accurate and libraries' implementations contain no bugs.

[3] http://bugs.sun.com/bugdatabase/

[4] http://findbugs.sourceforge.net/bugDescriptions.html

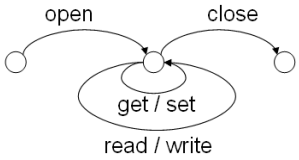

Figure 1. Abstract resource model

In summary, the inconsistencies among API documents, libraries' implementations, and programmers' understandings of libraries can introduce bugs. It is desirable to infer formal specifications from API documents and check the inferred specifications for bugs.

3 Resource Model

Abstract resource model. In this paper, we summarize the correct usages of a resource into abstract resource models (AR-model). Figure 1 shows an abstract resource model that is related to resources such as files, net resources, and devices. The model describe that before we use (get, set, read, and write) a resource, the resource needs to be opened, and after we use a resource, the resource needs to be closed. Programmers should follow the correct usages described in abstract resource model to avoid possible bugs such as memory leak. We latter use a triple $\langle acquire, use, release\rangle$ to denote an abstract resource model. For example, the abstract resource model shown in Figure 1 can be denoted as $\langle open, get/set\ read/write, close\rangle$

Action-resource templates. As shown in Section 1, many documents simply describe what kind of action the method takes against a particular resource in natural languages. NLP techniques such as phrase/clause parsing and Part-Of-Speech (POS) tagging can be used to parse documents for those resource-related ones. Figure 2 shows the parsed syntatic tree of the preceding document "*opens the device, indicating that ...*". In particular, phrase parsing can divide a sentence into syntactically correlated parts of words such as noun phrases and verb phrases. Clause parsing can recognize the word sequences of these phrases such as subjects and predicates. Besides that, POS tagging can further identify the POS tags for each word such as nouns, verbs, and determiners. Penn-II treebank's website[5] lists the details of these POS tags. Although existing NLP techniques still have difficult to understand the correct meanings of documents, the state-of-art can already achieve parsing and tagging accuracy of more than 90% on well written new articles [17].

For the documents of a method, the resource-related ones can be found out by matching the parsed syntatic tree with the action-resource templates as follows.

(1). Omitted subject (This method) $\langle verb\rangle$ + $\langle noun\rangle$.

(2). This method/It + $\langle verb\rangle$ + $\langle noun\rangle$.

[5] http://bulba.sdsu.edu/jeanette/thesis/PennTags.html

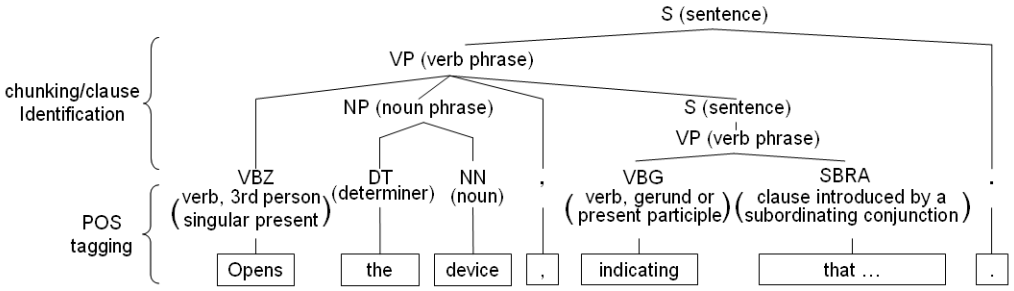

Figure 2. Syntactic tree

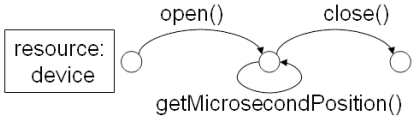

Figure 3. Concrete resource model

Most documents of methods under Javadoc's "*Method Summary*" follow the first template. There are also documents that follow the second template. For example, the document for `java.beans.XMLDecoder.close()` says that "*This method closes the input stream associated with this stream.*" For each sentence, its verb and corresponding parsed syntatic tree can be organized in pair for latter processes.

Concrete resource model. Concrete resource models (CR-model) describe the correct usages of concrete methods from libraries. To build CR-models, we need to refine the templates for actions and resources according to the extracted verbs. Synonym lexicons such as WordNet [8] can be used in the CR-models building process.

If the verbs are synonyms of open and close, the refined templates are:

(1). Omitted subject $\langle action \rangle$ + $\langle resource \rangle$.

(2). This method/It + $\langle action \rangle$ + $\langle resource \rangle$.

If the verbs are synonyms of get and set, the refined templates are:

(1). Omitted subject $\langle action \rangle$ + $\langle attribute \rangle$ + of/on/about + $\langle resource \rangle$.

(2). This method/It + $\langle action \rangle$ + $\langle attribute \rangle$ + of/on/about + $\langle resource \rangle$.

If the verbs are synonyms of read and write, the refined templates are:

(1). Omitted subject $\langle action \rangle$ + $\langle content \rangle$ + from/to + $\langle resource \rangle$.

(2). This method/It + $\langle action \rangle$ + $\langle content \rangle$ + from/to + $\langle resource \rangle$.

For example, the document of `javax.sound.midi.MidiDevice.getMicrosecondPosition()` says that "*Obtains the current time-stamp of the device, in microseconds*". As *obtain* is a synonym of *get*, the extracted action is "obtains", and the extracted resource is "device". The noun "time-stamp" is an attribute of the resource "device".

For a library, after its methods' actions and resources are extracted, these methods can be grouped by the same resource to build the corresponding CR-model of the resource. Figure 3 shows a built CR-model for J2SE's midi device. As CR-models are more formal than natural languages, these model can be used to check bugs discussed in Section 2 automatically.

4 Implementation

Figure 4 shows the overview of XXX. XXX is consist of a specification generator and a bug finder. In particular the specification generator of XXX takes HTML documents generated by Javadocs as inputs and infers specifications in the form of CR-models (Section 4.1). The bug check the inferred specifications against existing open source clients for bugs (Section 4.2).

4.1 Specification Generator

Html analyzer. Javadoc is an industry standard tool to generate API document for libraries in Java. It is widely used [9], and the generated documents are in structured HTML format. J2SE's API documents[6] show an example of such documents generated by Javadoc. For the methods of a library, their documents are listed item by item under the topics such as "*Method Summary*" and "*Method Detail*". As the documents under "*Method Summary*" are more concise than the documents under "*Method Detail*", for each method, XXX focuses on its documents under "*Method Summary*". As these HTML documents are structured, it is feasible to extract documents for every method automatically. In particular, XXX uses an HTML parser[7] to extract the documents of a method from API documents generated by Javadoc.

NLP analyzer. As documents are written in natural languages, they need to be parsed for latter process. For each sentence of extracted documents, XXX uses the Stanford parser [12] to identify the phrases/clauses and to tag POS for each word. The Stanford parser is open source and can

[6] http://java.sun.com/j2se/1.5.0/docs/api/

[7] http://htmlparser.sourceforge.net/

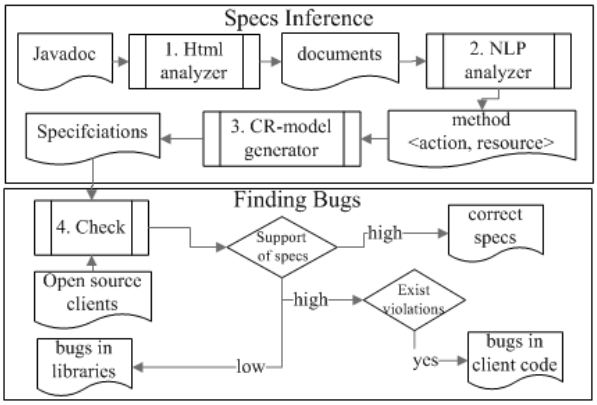

Figure 4. Overview

be downloaded from its website[8]. Like many other NLP tools, the Stanford parser has already trained in-house for standard natural languages such as English, so it needs no extra efforts for programmers to train the parser, although the Stanford parser does provide the capability to be trained for a particular language.

CR-model generator. For each method, after its documents are parsed into syntactic trees, the CR-model generator of XXX further extract the actions and resources from these trees. As the action and resource of each method are extracted, the CR-model generate further groups the methods by their resources and maps these methods to AR-model using the techniques introduced in Section 3. The current implementation of XXX supports the AR-models such as $\langle open, get/set\ read/write, close\rangle$, $\langle lock, get/set\ read/write,\ unlock\rangle$, $\langle acquire,\ get/set\ read/write, release\rangle$, and $\langle connect, get/set\ read/write, close\rangle$.

4.2 Bug Finder

5 Experiment

6 Discussion

7 Related Work

Mining specifications from clients. As clients contain many valuable usages of libraries, many approach are proposed to mine specifications from client code statically or dynamically. Based on their outputs, these mining approaches can be mainly divided into automata-based approaches and sequence-based approaches. For automata-based approaches, Strauss developed by Ammons et al. [2] uses an extended Angluin algorithm [21] to mine automata from the traces that are related by traditional dataflow dependencies. SMArTIC developed by Lo and Khoo [16] further improves Strauss by introducing clustering techniques to refine traces before the mining process. Kremenek et al. [13] use an automata template to find methods that fit the automata template. Whaley et al. [25] mine automata-like models from traces. Dallmeier et al. [5] present an approach to extract predicates and to mine named automata from traces. Cook and Wolf [4] reduce the general problem of mining automata from traces to the classical grammar inference problem. The problem has been proved to be NP-complete [10]. For sequence-based approaches, Eagler et al. [7] use the z-statistic value as a support value to mine frequent call sequences. Yang et al. [26] use method-call-pair templates to mine the method calls whose supports are greater than a threshold into frequent call sequences. Li and Zhou [14] use frequent itemset mining to extract implicit programming properties and detect their violations for detecting bugs. DynaMine developed by Livshits and Zimmermann [15] mines properties from software revision histories. Wasylkowski et al. [24] use sequence mining for frequent call sequences to detect anomalies in clients. Ramanathan et al. [22] use sequence mining for frequent call sequences from the statically extracted traces. These approaches use existing client code as an input. When a library is not popular or new, its clients can hardly be found. XX takes API documents as inputs, complementing these approaches.

Inferring specifications from library code. As many libraries provide source files, it is feasible to infer specifications from library code. Alur et al. [1] proposes an approach that combines Angluin's algorithem [3] with model checking to infer automata for a given component. Gowri et al. [11] propose a static approach to lean objects's relationships, internal states, and a first order specification. Due to the complexity of source files, these approaches are not testified on large scale libraries. XX infer specifications from documents and are testified on large scale libraries such as XX and XX.

Inferring specifications from documents. For most of open source and commercial libraries, their documents are all provided. It is possible to infer and check specifications from these documents although they are often written in natural languages. iComment developed by Tan et al. [23] is claimed to be the first work to infer specifications from documents. It infers specifications that are explicitly stated in documents and uses the inferred specifications for real bugs in projects such as Linux. XXX also infers specifications from API documents, but these specifications are not explicitly in documents, complementing Tan et al.'s work.

Checking specifications for bugs.

Empirical study of documents.

[8] http://nlp.stanford.edu/software/lex-parser.shtml

8 Conclusion

References

[1] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Notices*, 40(1):98–109, 2005.

[2] G. Ammons, R. Bodík, and J. Larus. Mining specifications. In *Proc. POPL*, pages 4–16, 2002.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[4] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[5] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Proc. WODA*, pages 17–24, 2006.

[6] S. de Souza, N. Anquetil, and K. de Oliveira. A study of the documentation essential to software maintenance. *Proc. International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, pages 68–75, 2005.

[7] D. Engler, D. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. SOSP*, pages 57–72, 2001.

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

[9] A. Forward and T. Lethbridge. The relevance of software documentation, tools and technologies: a survey. *Proc. ACM Symposium on Document Engineering*, pages 26–33, 2002.

[10] E. Gold. Complexity of automatic identification of given data. *Information and Control*, 10:447–474, 1978.

[11] M. Gowri, C. Grothoff, and S. Chandra. Deriving object typestates in te presence of inter-object references. In *Proc. Conference on Object-oriented programing, systems, languages, and applications*, pages 77–96, 2005.

[12] D. Klein and C. Manning. Accurate unlexicalized parsing. *Proc. Annual Meeting of the Association for Computational Linguistics*, pages 423–430, 2003.

[13] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proc. OSDI*, pages 259–272, 2006.

[14] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. ESEC/FSE*, pages 306–315, 2005.

[15] V. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proc. ESEC/FSE*, pages 31–40, 2005.

[16] D. Lo and S. Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proc. ESEC/FSE*, pages 265–275, 2006.

[17] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.

[18] D. Novick, E. Elizalde, and N. Bean. Toward a more accurate view of when and how people seek help with computer applications. *Proc. International Conference on Design of Communication*, pages 95–102, 2007.

[19] D. G. Novick and K. Ward. Why don't people read the manual? In *Proc. International Conference on Design of Communication*, pages 11–18, 2006.

[20] E. Perry, M. Sanko, B. Wright, and T. Pfaeffle. Oracle 9i JDBC developer's guide and reference. Technical report, http://www.oracle.com, Mar 2002.

[21] A. Raman and J. Patrick. The sk-strings method for inferring PFSA. In *Proc. Machine Learning Workshop Automata Induction, Grammatical Inference, and Language Acquisition*, 1997.

[22] M. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proc. ICSE*, pages 240–250, 2007.

[23] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or Bad Comments?*/. *Operating Systems Review*, 41(6):145, 2007.

[24] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. ESEC/FSE*, pages 35–44, 2007.

[25] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. ISSTA*, pages 218–228, 2002.

[26] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proc. ICSE*, pages 282–291, 2006.