Inferring specifications for resources from natural language API documentation

Hao Zhong · Lu Zhang · Tao Xie · Hong Mei

Abstract Many software libraries, especially those commercial ones, provide API documentation in natural languages to describe correct API usages. However, developers may still write code that is inconsistent with API documentation, partially because many developers are reluctant to carefully read API documentation as shown by existing research. As these inconsistencies may indicate defects, researchers have proposed various detection approaches, and these approaches need many known spec-

H. Zhong (✉)
Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing, China
e-mail: zhonghao@itechs.iscas.ac.cn

L. Zhang · H. Mei
School of Electronics Engineering and Computer Science, Peking University, Beijing, China

L. Zhang
e-mail: zhanglu@sei.pku.edu.cn

H. Mei
e-mail: meih@sei.pku.edu.cn

L. Zhang · H. Mei
The Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China

T. Xie
Department of Computer Science, North Carolina State University, Raleigh, USA
e-mail: xie@csc.ncsu.edu

ifications. As it is tedious to write specifications manually for all APIs, various approaches have been proposed to mine specifications automatically. In the literature, most existing mining approaches rely on analyzing client code, so these mining approaches would fail to mine specifications when client code is not sufficient. Instead of analyzing client code, we propose an approach, called Doc2Spec, that infers resource specifications from API documentation in natural languages. We evaluated our approach on the Javadocs of five libraries. The results show that our approach performs well on real scale libraries, and infers various specifications with relatively high precisions, recalls, and F-scores. We further used inferred specifications to detect defects in open source projects. The results show that specifications inferred by Doc2Spec are useful to detect real defects in existing projects.

Keywords Inferring specifications · API documentation

1 Introduction

Application Programming Interface (API) documentation plays an important role for developers to cope with software libraries. From API documentation, developers can find much useful information such as method descriptions, and learn how to correctly use libraries. For example, when developers use J2EE, they can refer to the J2EE's Javadoc[1] for its correct usages.

Still, developers may write code that is inconsistent with API documentation, partially because many developers are reluctant to carefully read API documentation, as shown by existing research on developers' behavior (Novick and Ward 2006). These inconsistencies often indicate defects, and researchers (Lu et al. 2008) point out that many defects are related to resource usages. To detect defects that are related to resource, researchers (*e.g.*, Igarashi and Kobayashi 2005) propose various approaches that analyze resource usages, and these approaches need specifications that describe correct usages of resources.

As API documentation contains much information on resource usages, it is feasible to infer resource specifications from API documentation although some usages are implicit. For example, the description of java.sql.ResultSet.deleteRow() is "*Deletes the current row from this ResultSet object and from the underlying database*", and the description of java.sql.ResultSet.close() is "*Releases this ResultSet object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed*". Although each description simply describes what kind of *action* a method takes on a particular *resource* and does not explicitly contain any rules, an experienced developer can extract an implicit specification deleteRow() → close()[2] from the preceding two descriptions. The specification describes that close() should be called if deleteRow() is already called since a used resource needs to be eventually closed. As another example, the description of

---

[1]http://java.sun.com/javaee/5/docs/api/.

[2]The semantics of a() → b() is that an invocation of the b() method should eventually follow an invocation of the a() method on the same object.

`javax.sound.midi.MidiDevice.open()` is "*Opens the device, indicating that it should now acquire any system resources it requires and become operational*", and the description of `javax.sound.midi.MidiDevice.close()` is "*Closes the device, indicating that the device should now release any system resources it is using*". Similarly, an experienced developer can extract an implicit specification `open()` → `close()`. The specification describes that `close()` should be called if `open()` is already called since an opened resource also needs to be eventually closed.

The preceding specifications are valuable to detect resource related defects. However, it is challenging to infer from API documentation due to two main factors: (1) it requires accurate linguistic analysis since API documentation is in natural languages; (2) it requires information from multiple method descriptions to be synthesized since resource usages are typically implied in descriptions of multiple methods.

In this paper, we propose a novel approach, called Doc2Spec, that infers resource specifications from existing API documentation in natural languages. As our approach does not need any source code from either libraries or their clients, it is capable of inferring specifications when source code is unavailable or insufficient. Thus, it complements existing approaches of mining specifications from source code (see Sect. 4 for details).

This paper makes the following main contributions:

– We propose a novel approach, called Doc2Spec, that uses a Natural Language Processing (NLP) technique to analyze natural language API documentation and infers resource specifications.
– We implemented a tool for Doc2Spec and conducted an evaluation on API documentation of five libraries. The results show that our approach performs well on real scale libraries and infers various specifications with relatively high precisions, recalls, and F-scores.
– We further conducted an evaluation to detect defects using inferred specifications. The results show that these specifications are useful to detect previously known or unknown defects in open source projects.

The remainder of our paper is as follows. Section 2 introduces the background of our approach. Section 3 illustrates our approach using an example. Section 4 presents related work. Section 5 presents the details of our approach. Section 6 presents our evaluations. Section 7 presents the benefits of our approach. Section 8 discusses the issues of our approach. Section 9 concludes.

2 Background

In this section, we introduce the background of our approach.

2.1 API documentation

It is a common practice to provide API documentation in natural languages for programmers. For example, Fig. 1 shows a screen snapshot of the API documentation

Fig. 1 API documentation of J2SE 6.0

provided by J2SE 6.0[3]. As a part of Java 2 SDK, Javadoc[4] is a tool that is able to generate API documentation in the HTML format from code comments for libraries in Java. In this paper, we also use "Javadoc" to denote API documentation generated by this tool. As required by an official guidance[5], Javadoc includes top-level specifications, package specifications, class/interface specifications, field specifications, and method specifications. In particular, each method specification may include its expected behavior, state transitions, ranges of valid argument values, null argument values, range of return values, defined algorithms, OS/hardware dependencies, allowed implementation variances, causes of exceptions, and security constraints.

When developers write Javadocs, they should follow some conventions and styles. For example, another official guidance[6] introduces some conventions for writing Javadocs. As shown in Fig. 2, the expected behavior of a method is described typically under the "Method Summary" topic or the "Constructor Summary" topic items by items. In this paper, we refer to the expected behavior of a method as the method description of the method. In most cases, method descriptions start with a verb phrase that describes specific actions against some resources. Besides method descriptions, generated Javadocs typically include inheritance relations among classes/interfaces as shown in Fig. 3.

API documentation besides Javadocs typically also includes method descriptions and inheritance relations. For example, Fig. 4 shows some captured contents from the MSDN library[7]. In MSDN, method descriptions are under the "Constructors" topic and the "Methods" topic, and inheritance relations are under the "Inheritance Hierarchy" topic. Our approach infers specifications from method descriptions and inheritance relations, and is general since API documentation typically includes method descriptions and inheritance relations.

---

[3]http://java.sun.com/javase/6/docs/api/.

[4]http://java.sun.com/j2se/javadoc/.

[5]http://java.sun.com/j2se/javadoc/writingapispecs/index.html.

[6]http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html.

[7]http://msdn.microsoft.com/en-us/library/ff361664(v=VS.100).aspx.

Fig. 2 Method descriptions

Fig. 3 Inheritance relations

2.2 Resource usage

There are many resources such as files, memories, and database connections in API libraries. When programmers use these resources, some rules should be followed. For example, a resource should be eventually closed after it is allocated. In particular, Fig. 5 shows the specification template proposed by Kremenek et al. (2006). In their specification template, *ro* denotes allocators of resources, and *co* denotes deallocators of resources. As described by the template, if programmers use a resource correctly, it leads to the *OK* state, and if not, it leads to the *Bug* state.

Our approach also uses a template as shown in Fig. 6 to describe resource usages. Instead of two types of methods defined in Fig. 5, our approach defines five types of methods: creation, lock, manipulation, unlock, and closure as follows.

Creation methods represent actions that create or return created resources (*e.g.*, *create*, *open*, and *connect*).

Lock methods represent actions that lock created resources (*e.g.*, *lock* and *acquire*).

Manipulation methods represent actions that manipulate created resources (*e.g.*, *get*, *set*, and various other actions).

Constructors

	Name	Description
⁼◆✖	ArrayList	Initializes a new instance of the ArrayList class that is empty and has the default initial capacity.
⁼◆✖	ArrayList (ICollection)	Initializes a new instance of the ArrayList class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied.
⁼◆✖	ArrayList (Int32)	Initializes a new instance of the ArrayList class that is empty and has the specified initial capacity.

Top

Methods

	Name	Description
⁼◆S	Adapter	Creates an ArrayList wrapper for a specific IList.
⁼◆✖	Add	Adds an object to the end of the ArrayList.

Inheritance Hierarchy

System.Object
  System.Collections.ArrayList
    System.Windows.Forms.DomainUpDown.DomainUpDownItemCollection

Fig. 4 Selected contents from MSDN
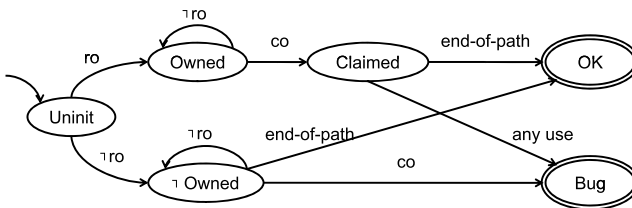
Fig. 5 Specification template proposed by Kremenek *et al.*

Fig. 6 Specification template defined in our approach
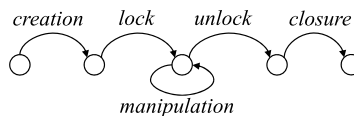
Unlock methods represent actions that unlock locked resources (*e.g.*, *unlock* and *release*).

Closure methods represent actions that release created resources (*e.g.*, *destroy*, *close*, and *free*).

The creation methods of our template are equal to the *ro* methods in Fig. 5, and the closure methods of our template are equal to the *co* methods in Fig. 5. Different

from the template in Fig. 5, our template does not show the transitions that lead to defects explicitly. In Sect. 2.3, we further discuss caused defects when programmers do not follow our template to use resources.

2.3 Related defects

Defects may be introduced in client code if programmers do not follow our template to use resources. We summarize some types of defects as follows.

Defect 1: In this type of defects, client code manipulates a resource without creating the resource. This type of defects will lead to exceptions if the corresponding code is executed once.

Defect 2: In this type of defects, client code creates resources but never closes resources. This type of defects will exhaust resources if the corresponding code is executed many times.

Defect 3: In this type of defects, client code locks resources but never unlocks resources. This type of defects will lead to deadlock if the corresponding code is executed once.

Defect 4: Some resources such as `resultset` in Java as described in Sect. 1 do not have explicit creation methods, and these resources are still necessary to be closed after they are manipulated. In this type of defects, client code manipulates resources but never closes resources. This type of defects will exhaust resources if the corresponding code is executed many times.

To detect the preceding types of defects, our approach defines the following criteria for resources correspondingly:

Cr/M: A manipulated resource should be already created if a resource has manipulation methods and creation methods.

Cr/Cl: A created resource should eventually be closed if a resource has creation methods and closure methods.

L/U: A locked resource should eventually be unlocked if a resource has lock methods and unlock methods.

M/Cl: A manipulated resource should be closed eventually if a resource has manipulation methods and closure methods. This criterion is necessary because a resource may not have creation methods. In such a circumstance, the criteria *Cr/M* and *Cr/Cl* are not applicable.

3 Example

In this section, we use a resource in J2EE named `CCIConnection` to illustrate the main steps of our approach and how to use the inferred specification to detect defects.

3.1 Inferring specifications

Our approach consists of three main steps to infer specifications from API documentation.

The first step is to extract method descriptions and class/interface hierarchies from API documentation. In this example, from its Javadoc[8], our approach extracts three method descriptions of the `javax.resource.cci.Connection` interface as follows:

`createInteraction()`: "*Creates an interaction associated with this connection.*"

`getMetaData()`: "*Gets the information on the underlying EIS[9] instance represented through an active connection.*"

`close()`: "*Initiates close of the connection handle at the application level.*"

The second step is to build an action-resource pair from each method description. For a method, its action-resource pair denotes what *action* the method takes on what *resource*. In this example, our approach builds the action-resource pairs for the three methods as follows:

`createInteraction()`:⟨*create*, *connection*⟩.

`getMetaData()`:⟨*get*, *connection*⟩.

`close()`:⟨*close*, *connection*⟩.

As method descriptions are written in natural languages, it is difficult to define simple templates to extract action-resource pairs. In particular, the actions of `createInteraction()` and `getMetaData()` are predicate verbs, whereas the action of `close()` is an accusative object. Although the resources of all the three methods are preposition objects, there are multiple preposition objects in one description, and the locations of these resources are different. Here, if we simply pick those common concrete nouns as resources, we may mix specifications of different resources and infer false specifications (see Sect. 8.1 for details). Our approach leverages an NLP technique to extract action-resource pairs accurately (see Sect. 5.2 for details).

The final step is to infer automata for resources based on action-resource pairs and class/interface hierarchies. First, for each class/interface, our approach groups methods into categories according to resource names and class/interface hierarchies. In this example, the three methods are grouped into one category since their resources are of the same name and the three methods are declared by the same interface. Second, in each category, our approach maps methods to different types according to their actions. In this example, our approach maps the three methods to their types as follows:

`createInteraction()` → a creation method.

`getMetaData()` → a manipulation method.

`close()` → a closure method.

Finally, in each category, our approach builds an automaton based on our predefined specification template shown in Fig. 6. Figure 7 shows the inferred specification for the resource of interest. Our approach tailors our specification template to build the automaton shown in Fig. 7 (see the end of Sect. 5.3 for details).

---

[8]http://java.sun.com/j2ee/apidocs-1_5-fr/javax/resource/cci/Connection.html.

[9]Enterprise Information System (EIS).

Fig. 7 Resource specifications
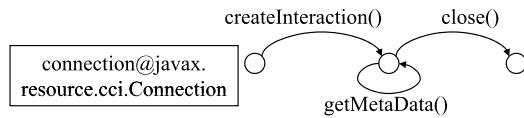
3.2 Detecting defects

To confirm the usefulness of an inferred specification, we need to investigate whether we can detect defects from violations of the specification. In this example, we can check whether `close()` is eventually called in all possible execution paths of a code snippet for violations of the specification as shown in Fig. 7. In fact, we did find such a violation in a code snippet as follows:

```
public float getHomeEquityRate(){
  ...
  try{
     javax.resource.cci.Connection myCon
        = connFactory.getConnection();
     javax.resource.cci.Interaction interaction
        = myCon.createInteraction();
     ...
     myCon.close();
     ...
  }catch(Exception e){
     e.printStackTrace();
  }
}
```

In this code snippet, a local variable named `myCon` is not closed in the *exception* clause. The violation is determined as a suspected defect since unclosed connections may cause memory leaks. Such a case shows that the specification shown in Fig. 7 is useful to detect defects. Here, although Java has a garbage-collector-enabled platform, unclosed connections may still cause memory leaks. For example, the Oracle 9i JDBC Developer's Guide and Reference (Perry et al. 2002) warns "*If you do not explicitly close your ResultSet and Statement objects, serious memory leaks could occur*". The work proposed by Xu and Rountev (2008) also focuses on memory leaks in Java programs.

4 Related work

In this section, we introduce related approaches and discuss their differences from our approach.

4.1 Mining specifications

As client projects contain many valuable usages of libraries, many approaches have been proposed to mine specifications from client code statically or dynamically. These mining approaches can be divided into automata-based approaches and sequence-based approaches based on their outputs.

For automata-based approaches, Ammons et al. (2002) proposed an approach and its supporting tool *Strauss* that uses an extended k-tails algorithm (Raman and

Patrick 1997) to mine automata from execution traces that are related by traditional dataflow dependencies. Lo and Khoo (2006) improved *Strauss* by introducing clustering techniques to refine traces before the mining process. Whaley et al. (2002) mined automata-like models from execution traces and refine these models using code analysis. Kremenek et al. (2006) used Bayesian learning to match methods with a predefined automata template for specifications. Alur et al. (2005) used randomly generated test cases as clients and use Angluin's algorithm (Rivest and Schapire 1993) to infer automata from call sequences that do not throw exceptions. Gowri et al. (2005) also used a client emulator as clients, and their approach uses some analysis results such as objects' relationships, internal states, and their specifications of libraries. Gabel and Su (2008) proposed a symbolic algorithm based on binary decision diagrams to mine automata from execution traces. Gabel and Su (2010) proposed an approach that mines automata and detects anomalies online. Lee et al. (2011) apply a trace slicer before mining, and thus their approach is able to mine parametric specifications in the form of automata.

For sequence-based approaches, Engler et al. (2001) used the Z-statistic value as the support to mine frequent call pairs from source files. Yang et al. (2006) proposed an algorithm to mine call pairs from execution traces. Weimer and Necula (2005) proposed an approach to mine and filter method pairs from execution traces. Li and Zhou (2005) used frequent itemset mining to extract implicit programming properties and detect their violations for detecting bugs. Livshits and Zimmermann (2005) proposed an approach of mining properties from software revision histories. Wasylkowski et al. (2007) used frequent sequence mining to mine frequent call sequences from clients and use anomalies to detect bugs in client code. Ramanathan et al. (2007) used frequent sequence mining to mine frequent call sequences from execution traces extracted statically in client code. Zhong et al. (2008a) proposed an early filtering approach to improve accuracies of existing approaches. Zhong et al. (2009a) proposed an approach that combines clustering with sequence mining to mine context-sensitive specifications from client code. Acharya et al. (2007) proposed an approach that mines partial orders of API methods from client code. Acharya and Xie (2009) proposed an approach that mines error-handling specifications for procedural languages such as Thummalapenta and Xie (2009b) proposed an approach that mines specifications combining frequent call sequences and return values. Thummalapenta and Xie (2009a) proposed an approach that mines specification particularly for handling exceptions. Gabel and Su (2010) proposed an approach that infers and enforces specifications from execution history online. Lo and Maoz (2010) combined sequence mining with invariant mining to mine live sequence charts.

Most of these preceding previous approaches use existing client code as an input. When a library is not popular or new, its clients are difficult to find, and randomly generated test cases may not reflect real usages of libraries. Our previous work (Thummalapenta and Xie 2008) shows that even in a popular library, some methods or classes are rarely used. As our approach does not need client code, it is able to infer specifications when these methods have descriptions, complementing these previous approaches. These inferred specifications can be used to detect bugs when developers leverage libraries to develop client code.

Arnout and Meyer (2003) proposed an approach to manually extract contracts from .NET documents. Their inferred contracts consist of invariant-like precondi-

tions and postconditions and do not capture legal call sequences as our approach does. Tan et al. (2007) proposed *iComment* that extracts rules from rule-containing comments in source files. One such comment is the comment of free_irq() in the Linux kernel: "*This function must not be called from interrupt context*". *iComment* first identifies rule-containing comments using a trained decision tree and then uses a set of templates to infer rules from these comments. In this example, the corresponding template is "$\langle F_A \rangle$ *must (NOT) be called from* $\langle F_B \rangle$". *iComment* further uses extracted rules to find inconsistencies between comments and code. These rules can be considered as specifications. Although both *iComment* and our approach infer specifications from texts in natural languages, our approach differs from *iComment* as follows. First, our approach focuses on inferring resource specifications from API documentation, whereas *iComment* focuses on inferring general specifications from comments. Second, each specification inferred by our approach is implicit in multiple textual descriptions, whereas each specification inferred by *iComment* is explicit in one sentence.

4.2 Natural language analysis in software engineering

As many documents in software engineering are written typically in natural languages, it is feasible to leverage NLP techniques to analyze these documents.

Researchers have used various NLP techniques to analyze requirement documents. Ambriola and Gervasi (1997) implemented a set of tools to aid gathering, selecting, and validating requirements. These tools use various NLP techniques such as tagging, synonym analysis, and anaphora analysis. Goldin and Berry (1997) implemented AbstFinder to infer abstractions from requirements. To parse requirements, AbstFinder uses NLP techniques such as stemming and tagging. Sawyer et al. (2002) used part-of-speech (POS) and semantic tagging to support requirement synthesis from documents. Fantechi et al. (2003) used syntactic parsing to analyze use cases from requirement documents. Shepherd et al. (2007) used various NLP techniques such as stemming and POS tagging to locate and understand action-oriented concerns. Gervasi and Zowghi (2005) implemented CARL that detects inconsistencies in requirements. To parse requirements in natural languages, CARL uses various NLP techniques such as tagging and morphosyntactic analysis. Kof (2007) used POS tagging to identify missing objects and actions in requirement documents.

Researchers have used various NLP techniques to analyze bug reports and defect reports. Gegick et al. (2010) used a prediction model to identify security bug reports from other bug reports. Anvik et al. (2006) used support vector machines to mine descriptions of bug reports, and predicted the developer who should deal with a given bug report. Jeong et al. (2009) used the Markov chain model to analyze bug reports to predict who should fix a bug report. Runeson et al. (2007) used NLP techniques such as tokenization, stemming, vector space representation, and similarity calculation to detect duplicated defect reports. Wang et al. (2008) further combined text analysis with execution information of failed test cases to detect duplicated bug reports. Dag et al. (2005) proposed a vector-space model to illustrate customer withes and requirements, and use the Cosine measure to link customer withes with corresponding requirements. Hayes et al. (2006) used various information retrieval techniques such as latent semantic indexing to link requirements with code.

Our approach uses NLP techniques to infer specifications from API documentation, and API documentation is quite different in contents and structures from other documents such as requirement documents and bug reports. In addition, our approach first introduces a complicated NER technique to analyze documents, whereas the preceding approaches do not use such an NLP technique.

4.3 Improving documents

Jeong et al. (2009) conducted an empirical study on documentation of eSOA APIs, and presented implications for better documentation design. Robillard and DeLine (2011) conducted an empirical study to understand obstacles to learn APIs, and present many implications to improve API documentation. Shi et al. (2011) conducted an empirical study on API documentation evolution, and their findings are valuable for improving API documentation. Buse and Weimer (2010, 2008) presented various automatic techniques for exception documentation and synthesizing documentation for arbitrary programme differences across versions. Meziane et al. (2008) proposed an approach to generate specifications in natural languages from UML class diagrams. Dekel and Herbsleb (2009a, 2009b) proposed eMoose that pushes and highlights those rule-containing sentences from API documentation for developers. Stylos et al. (2009) displayed commonly used classes to help explore API documentation. Sridhara et al. (2010) proposed an approach that infers comments of Java methods from API code. Würsch et al. (2010) proposed an approach that supports programmers with natural language queries. Horie and Chiba (2010) proposed an extended Javadoc tool that provides new tags to maintain crosscutting concerns in documentation.

Their approach improves the quality of documentation, whereas our approach infers specifications from API documentation and detects bugs in code.

5 Approach

The overview of our approach is shown in Fig. 8. Our approach focuses on API documentation in the form of Javadoc. Figures 2 and 3 show several pieces of J2SE's Javadoc. We next present detailed steps of our approach.

5.1 Javadoc analysis

The first step of our approach is to extract method descriptions and class/interface hierarchies from Javadocs. As shown in Fig. 2, in Javadocs, method descriptions

Fig. 8 Overview of our approach
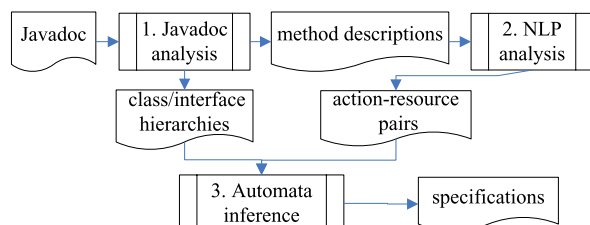
are under the "*Constructor Summary*" topic and the "*Method Summary*" topic, and class/interface hierarchies are under the "*Class Hierarchy*" topic and the "*Interface Hierarchy*" topic. As Javadocs are in a structured HTML format, these topics are easy to locate. Specifically, for each method in a class, our approach locates the "*Constructor Summary*" topic and the "*Method Summary*" topic by searching for the "*constructor_summary*" anchor name and the "*method_summary*" anchor name. Some methods may have no descriptions, and our approach ignores these methods (see Sect. 8 for details). To extract class/interface hierarchies, our approach first searches for the file named "*package-tree.html*" in the directory of each package, and then locates the "*Class Hierarch*y" topic and the "*Interface Hierarchy*" topic through text matching.

5.2 NLP analysis

The second step of our approach is to build an action-resource pair from each method's description through NLP analysis. In the research field of NLP, the problem of identifying words belonging to a predefined category in a document is known as Named Entity Recognition (NER) (Chinchor 1997). In the literature, researchers have proposed rule-based approaches, dictionary-based approaches, and machine-learning-based approaches to recognize those entities. In particular, rule-based approaches (*e.g.*, Mikheev et al. 1999) use hand-crafted rules. A typical application of rule-based approaches is to recognize email addresses where entities are clearly defined through capital letters, symbols, and digits. Dictionary-based approaches (*e.g.*, Cohen and Sarawagi 2004) use a large collection of names as a dictionary for entities. A typical application of dictionary-based approaches is to recognize baseball players where a baseball site[10] has a list of all players. Machine-learning-based approaches (*e.g.*, Zhou and Su 2001) use mature machine learning techniques and various characteristics (*e.g.*, capitalization, digitalization, and contexts) for recognition.

As it is difficult to build hand-crafted rules or dictionaries for actions and resources, we choose machine-learning-based approaches for actions and resources. In particular, our approach uses the NER based on Hidden Markov Model (HMM) since it is reported to perform better than other machine-learning-based approaches (Zhou and Su 2001).

In NER, HMM is a five-tuple $\{\Omega_s, \Omega_o, \pi, A, B\}$ where

– $\Omega_s = \{s_1, \ldots, s_n\}$ is the finite set of states. In our approach, these states include *action*, *resource*, and *other*.
– $\Omega_o = \{o_1, \ldots, o_n\}$ is the set of observations. In our approach, Ω_s and Ω_o have one-to-one relations, and $o_i = \langle w_i, f_i \rangle$ where w_i is a word and $f_i = \langle F_i^W, F_i^M, F_i^{POS} \rangle$. Here, F^W denotes the word feature such as capitalization and digitalization; F^M denotes the morphological feature such as prefix and suffix; F^{POS} denotes the part-of-speech feature such as nouns, verbs, prepositions, adverbs, and adjectives.
– $\pi \in \Omega_s$ is the initial state. In our approach, π denotes the state of the first word of each method description.

---

[10]http://mlb.com.

Algorithm 1: Baum-Welch Algorithm

Input: O is the training data; ε is the threshold.
Output: λ is the trained HMM model.
begin

    $\lambda_0 \leftarrow \{\Omega_s, \Omega_o, \pi, A_0, B_0\}$;

    $\lambda \leftarrow \lambda_0$;

    repeat

        $\lambda' \leftarrow \lambda$;

        $\lambda \leftarrow compute(O, \lambda)$

    until $|\log P(O|\lambda) - \log P(O|\lambda')| < \varepsilon$;

end

- $A : \Omega_s \times \Omega_s \rightarrow [0, 1]$ is the probability distribution on state transitions. For example, $A(action, resource)$ denotes the probability of a transition from *action* to *resource*.
- $B : \Omega_s \times \Omega_o \rightarrow [0, 1]$ is the probability distribution on state symbol emissions. For example, $B(action, \langle close, f \rangle)$ denotes the probability of observing $\langle close, f \rangle$ when it is in the state *action*.

Our approach first uses the Baum-Welch algorithm (Baum et al. 1970) as shown in Algorithm 1 to train the parameters (A and B) from manually tagged method descriptions. In Algorithm 1, λ_0 is estimated by experiences. Given the training data with m tagged method descriptions ($O_1 O_2 \ldots O_m$) and an HMM model (λ), in each iteration, the new parameters (A and B) are calculated as follows:

$$\hat{a_{ij}} = \frac{\sum_{t=1}^{m} \gamma_{ij}(t)}{\sum_{t=1}^{m} \sum_k \gamma_{ik}(t)} \tag{1}$$

$$\hat{b_{jk}} = \frac{\sum_{t=1,o_t=v_k}^{m} \sum_l \gamma_{jl}(t)}{\sum_{t=1}^{m} \sum_l \gamma_{jl}(t)} \tag{2}$$

In (1) and (2), $\gamma_{ij}(t)$ is calculated as follows:

$$\gamma_{ij}(t) = \frac{a_i(t-1)a_{ij}b_{jk}\beta_j(t)}{P(O|\lambda)} \tag{3}$$

In (3), $\beta_i(t)$ is calculated as follows:

$$\beta_i(t) = \begin{cases} 0 & s_i(t) \neq s_0(t) \text{ and } t = m \\ 1 & s_i(t) = s_0(t) \text{ and } t = m \\ \sum_j \beta_j(t+1)a_{ij}b_{jk}O_{t+1} & \text{otherwise} \end{cases} \tag{4}$$

The training process as shown in Algorithm 1 builds a model with two parameters (A and B) to describe characteristics of actions and resources. After training, our approach then uses the Viterbi algorithm (Viterbi 1967) as shown in Algorithm 2 to

---

Algorithm 2: Viterbi Algorithm

Input: λ is the trained HMM model; $o_1 o_2 \ldots o_m$ is a method description.
Output: $s_1 s_2 \ldots s_m$ are the corresponding tags;
        $score_1 score_2 \ldots score_m$ are the corresponding scores.
begin
    for $i = 1$ **to** m **do**
        $\delta_1(i) \leftarrow \pi_i b_i(o_1)$;
        $\psi_1(i) \leftarrow 0$;
    $n \leftarrow |\lambda.\Omega_s|$;
    for $t = 2$ **to** m **do**
        for $j = 1$ **to** n **do**
            $\delta_t(j) \leftarrow \max_{1 \leq i \leq n} [\delta_{t-1} a_{ij}] b_j(o_i)$;
            $\psi_1(i) \leftarrow \arg \max_{1 \leq i \leq n} [\delta_{t-1}(i) a_{ij}]$;
    $s_m \leftarrow \arg \max_{1 \leq i \leq n} [\delta_m(i)]$;
    $score_m \leftarrow \delta_m(s_m)$;
    for $t = m - 1$ **to** 1 **do**
        $s_t \leftarrow \psi_{t+1}(s_{t+1})$;
        $score_t \leftarrow \delta_t(s_t)$;
end

---

tag method descriptions with scores based on the trained model, and Algorithm 2 uses the two trained parameters (*A* and *B*). In one method description, more than one word may be tagged as actions or resources. Our approach chooses the action and the resource both with the highest scores to build the action-resource pair for the method. Here, descriptions of some methods may not contain actions and resources. One such description is "*This method is not supported in the RtfWriter*". Our approach does not tag actions and resources for these descriptions since no words in these descriptions have common characteristics of actions or resources. Our approach does not build action-resource pairs for these methods and ignores them in the third step.

5.3 Automata inference

The final step of our approach is to infer automata for resources from action-resource pairs and class/interface hierarchies. For each class/interface, our approach first groups methods declared by the class/interface or the class/interface's super-classes/superinterfaces into categories since a class and its superclass may access one resource. In each category, resources of methods are of the same name. For example, Fig. 9 shows an interface hierarchy involving three interfaces. Our approach builds the action-resource pairs from the method descriptions as follows:

```
java.nio.channels.Channel.close()
```
   ⟨*close, channel*⟩ ←"*Closes this channel.*"
```
java.nio.channels.GatheringByteChannel.write()
```

Fig. 9 Hierarchical tree
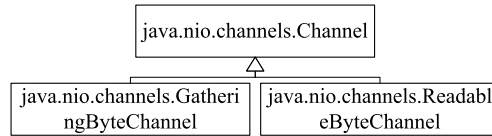
$\langle write, channel \rangle \leftarrow$ *"Writes a sequence of bytes to this channel from the given buffers."*

```
java.nio.channels.ReadableByteChannel.read()
```

$\langle read, channel \rangle \leftarrow$ *"Reads a sequence of bytes from this channel into the given buffer."*

Here, "buffer" is not recognized as a resource because its score is lower than "channel" in these two descriptions. Our approach next groups the methods for the three interfaces as follows:

```
java.nio.channels.GatheringByteChannel
  {write(),close()}
java.nio.channels.ReadableByteChannel
  {read(),close()}
java.nio.channels.Channel
  {close()}
```

Our approach puts these methods into categories based on their resource names and interface inheritances. Our approach does not group `read()` and `write()` into one category since their declaring interfaces are not subinterface and superinterface. To distinguish resources in different categories, we use "*resource name@class/interface name*" to denote the resource of one category for a class/interface.

After grouping, our approach further maps methods in each category to our predefined types according to their actions. If a method's action is within the representative actions, our approach simply maps the method to the type. Otherwise, our approach maps the method by synonyms of its action using a synonym dictionary named *Word-Net* (Fellbaum et al. 1998). If using synonyms still fails to resolve a method's action, we map the method into a manipulation method since most methods are manipulation methods. For example, as shown by its document[11], the `java.io.InputStream` class has 9 methods. Among these methods, only one method (*i.e.*, the `close()` method) is a closure method, and all the other methods are manipulation methods.

Our approach then builds an automaton for each category based on our predefined specification template (see Sect. 8.3 for the discussion on extensions of the specification template). In each category, our approach associates methods of each type to the type's corresponding transition in the specification template. In practice, some resources may have no methods of specific types and their automata need to be tailored. Our approach deletes transitions without any associated methods from our template and merges corresponding states. In the example shown in Sect. 3, as the resource has no *lock* methods, our approach deletes the transition labeled with "*lock*" and merges the exiting state and the entering state of the transition into one state. Our approach

---

[11] http://download.oracle.com/javase/1.5.0/docs/api/java/io/InputStream.html.

Table 1 Subjects

Library	Version	# Method	# Description
J2SE	5.0	25675	23829
J2EE	5.0	5670	5611
JBoss	4.0.5	26053	13869
iText	2.1.3	5846	4299
Oracle	10.1.0.5	2140	1916
Total		65384	49524

also deletes the transition labeled with "*unlock*" and merges the corresponding states since the resource has no *unlock* methods either. Thus, our approach builds the automaton shown in Fig. 7 from the specification template shown in Fig. 6. Here, some resources have only one type of method, and their inferred automata have only one state consequently. Our approach discards these automata since they are not helpful to detect defects.

6 Evaluations

We implemented a tool for our approach and conducted a series of evaluations using the tool. Our evaluations focus on three research questions as follows:

1. Can our approach perform well on real scale libraries (Sect. 6.1)?
2. What is the quality of inferred specifications (Sect. 6.2)?
3. Can inferred specifications be useful to detect defects (Sect. 6.3)?

In our evaluations, we manually tagged actions and resources for the descriptions of 687 methods in the J2SE Javadoc in one day, and trained Doc2Spec using these tagged descriptions in about ten seconds. We then used the trained Doc2Spec to infer resource specifications for five libraries as shown in Table 1. In Table 1, Column "*Library*" lists the names of the five used libraries. In the rest of the paper, we use "*Oracle*" to denote *Oracle JDBC driver*. For each library, column "*# Method*" lists the number of methods. Column "*# Description*" lists the number of methods with descriptions. We notice that each library has some methods without descriptions. Although only a small percentage of the total methods do not have descriptions in J2SE, J2EE, and Oracle, there are many methods without descriptions in JBoss and iText. We further discuss the impact of methods without descriptions on our approach in Sect. 8.3.4. We conducted all the evaluations on a PC with an Intel Pentium 2.26 GHz CPU and 1512M memory running Windows 2000 professional.

6.1 RQ1: Can our approach perform well on real scale libraries?

To evaluate the first research question, we used our implemented tool to infer specifications from the Javadocs of all the libraries shown in Table 1. Figure 10 shows six example inferred specifications. In a specification, the text box shows the resource, and the automaton shows the call relationship of the related methods. Each inferred specification describes some actions against a resource.
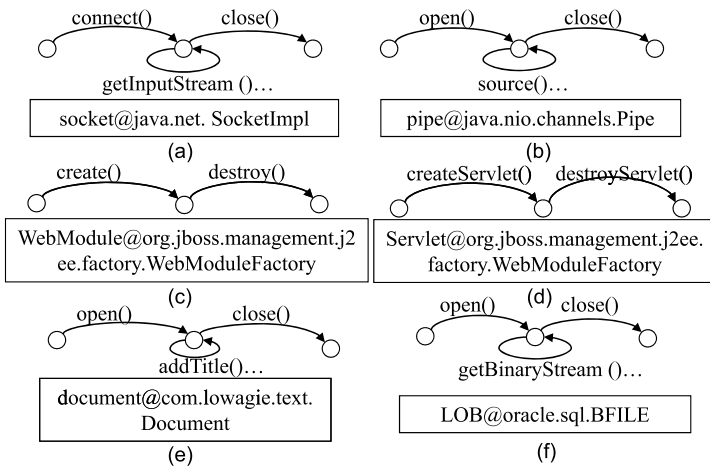
Fig. 10 Example inferred specifications

Table 2 Top ten actions

Rank	J2SE	J2EE	JBoss	iText	Oracle
1	**return**	**create**	**return**	**check**	**return**
2	**create**	**return**	**create**	add	**get**
3	determine	add	**get**	**get**	**close**
4	**get**	**get**	**check**	**return**	retrieve
5	retrieve	set	destroy	change	**create**
6	extract	**close**	**close**	**create**	execute
7	set	write	start	set	register
8	insert	start	connect	remove	allow
9	**check**	stop	stop	write	write
10	**close**	**check**	lock	**close**	**check**

For the inferred specifications, Table 2 shows the top ten actions, and Table 3 shows the top ten resources. In the two tables, columns "*Rank*" list ranks by frequencies of actions/resources, and columns of library names list corresponding actions/resources. In Table 2, we show the top ten actions that exist in all the five libraries with bold fonts. We find that the five actions such as *return*, *create*, *get*, *check*, and *close* exist in all the five libraries. From our results, methods typically provide similar actions even when methods are from different libraries. As actions are limited in number, limited resource templates may be sufficient to describe actions against resources. We also tried to find similar resources across libraries, but from the results of Table 3, resources are relatively complicated. Some resources such as *connection*, *element*, and *stream* exist in most of the five libraries, but we cannot find even one top-ten resource that exists in all the five libraries. It seems that resources are specific to libraries and thus are challenging for analysis. We further discuss issues on resources in Sect. 8.1.

Table 3 Top ten resources

Rank	J2SE	J2EE	JBoss	iText	Oracle
1	object	element	element	object	connection
2	DynAny	connection	field	document	statement
3	objectImpl	folder	connection	element	datasource
4	stream	transaction	node	stream	clob
5	context	consumer	queue	font	transaction
6	node	service	message	section	cache
7	document	context	client	content	blob
8	socket	stream	service	phrase	lob
9	channel	message	servlet	profile	datum
10	element	producer	module	node	file

Fig. 11 Percentages of specifications that involve specific numbers of methods
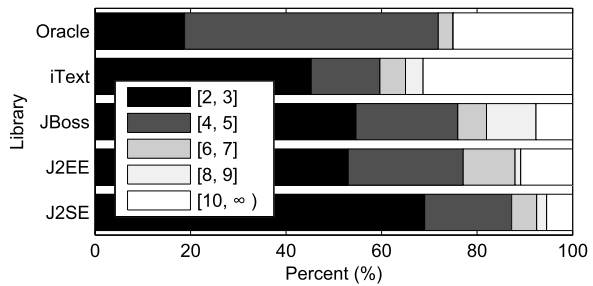
Fig. 12 Percentages of specifications that involve specific numbers of classes/interfaces
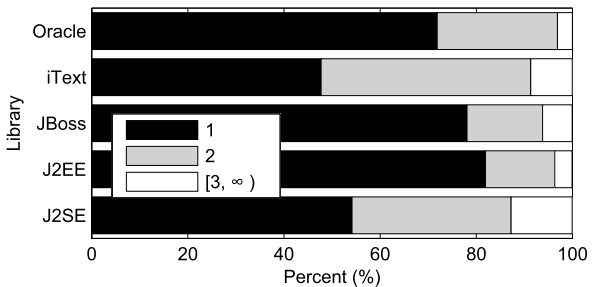
To analyze the distribution of specifications in each library, we further present Figs. 11 and 12. In the two figures, the vertical axes show the names of the libraries, and horizontal axes show percentages of specifications that involve specific numbers of methods or classes/interfaces. For example, the black bar of "J2SE" in Fig. 11 shows that 69.1% of the specifications inferred from the Javadoc of J2SE have 2 or 3 methods. Overall, the results indicate that our approach is able to infer various and complex specifications, although most of the inferred specifications involve only one or two classes/interfaces and fewer than five methods.

When our tool inferred specifications, we recorded related data during documentation analysis and specification inference for each library, and Table 4 shows the

Table 4 Performance of
Doc2Spec

Library	# Spec	Doc Time	Spec Time
J2SE	3250	0.12	0.73
J2EE	83	1.15	1.82
JBoss	373	1.15	0.38
iText	243	0.46	0.22
Oracle	32	0.88	0.70
Total	3981	0.27	0.68

results. In Table 4, column "*#Spec*" lists the number of inferred resource specifications. Column "*Doc Time*" lists average times used to extract method descriptions and class/interface hierarchies for each specification in seconds. Column "*Spec Time*" lists average times used to infer specifications based on the extracted information for each specification in seconds. Row "*Total*" lists total numbers for these columns.

From the results in Table 4, we have the following observations. First, for all the five libraries, both the time used to extract method descriptions and the time used to infer specifications are acceptable, and it takes less than one second to infer a specification on average. Second, the time used to infer specifications of J2SE is much longer than the time of other libraries. We suspect the reason to be that there are much more inferred specifications for J2SE. Finally, for each library, the time used to extract method descriptions is largely proportional to the number of methods in the library. This observation indicates that extraction of method descriptions and class/interface hierarchies in Doc2Spec is scalable.

6.2 RQ2: what is the quality of inferred specifications?

To further investigate whether inferred specifications are accurate, we compared these inferred specifications with a golden standard. To prepare a golden standard for each library, we first grouped the class/interface hierarchies in the library, so that the hierarchies in each group are of the same maximum depth of inheritance. For each library, we then randomly selected one hierarchy from each group and manually built resource specifications for all the classes/interfaces within these selected hierarchies based on manually reading their Javadocs.

Table 5 shows the results. Column "*# Hierarchy*" lists numbers of selected hierarchies in each library. Column "*# Spec*" lists numbers of manually built specifications for each selected hierarchy. In statistical classification (Olson 2008), *Precision* for a category is the number of true positives divided by the total number of items labeled as belonging to the positive category, *Recall* is the number of true positives divided by the total number of items that actually belong to the positive category, and *F-score* is the weighted harmonic mean of *Precision* and *Recall*. In our comparison, *Precision*, *Recall*, and *F-score* are defined as follows.

$$Precison = \frac{true\ positives}{true\ positives + false\ positives} \tag{5}$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{6}$$

Table 5 Precisions, Recalls, and F-scores of inferred specifications

Library	# Hierarchy	# Spec	Precision	Recall	F-score
J2SE	8	41	80.2%	82.2%	81.2%
J2EE	7	30	70.7%	79.3%	74.8%
JBoss	8	37	81.5%	74.0%	77.6%
iText	6	22	86.5%	85.2%	85.8%
Oracle	5	17	82.3%	86.2%	84.2%

$$F\text{-}score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{7}$$

In the preceding formulae, true positives represent those transitions that exist in both the inferred specifications and the golden standard; false positives represent those transitions that exist in the inferred specifications but not in the golden standard; false negatives represent those transitions that exist in the golden standard but not in the inferred specifications. We choose to calculate these statistical values by transitions since each transition can be used to detect defects instead of a whole specification.

The results show that our approach achieves reasonable precisions, recalls, and F-scores on all these libraries. The results also show that to infer specifications from a library, our approach does not require the training corpora to be from the same library because our approach achieves similar precisions, recalls, and F-scores for all the five libraries although we tagged method descriptions of only J2SE as the training corpora. A potential explanation lies in that most developers of API documentation follow a similar style to write Javadocs of libraries. Consequently, users of our approach can rely on a set of universal training corpora to deal with Javadocs instead of taking on the burden of preparing training corpora before taking advantage of our approach.

In summary, the statistics of resource specifications inferred by Doc2Spec show that our approach is able to infer various resource specifications from Javadocs of the five libraries and our approach is able to achieve relatively high precisions, recalls, and F-scores on specification inference. Still, we agree that it is quite important to further improve our approach for practical usages. In Sect. 8.2, we discuss issues to reduce false positives of our approach.

6.3 RQ3: can inferred specifications be useful to detect defects?

This evaluation aims to investigate whether programmers violate inferred specifications and thus introduce defects in true practice, so we used the inferred specifications to detect defects in open source projects.

6.3.1 Defect detection infrastructure

We implemented an infrastructure to automate this evaluation. For a specification, our infrastructure first searches for the specification's related code snippets from the

Table 6 Results of found violations

Library	API clients	Violations			
		Cr/Cl	*L/U*	*M/Cl*	*Cr/M*
J2SE	46	32	3	47	0
J2EE	23	8	2	32	0
JBoss	45	27	62	101	0
iText	9	11	0	4	0
Oracle	15	20	0	34	0
Total	138	383			

Internet using Google code search engine[12] (GCSE) and downloads these code snippets to local directories. For a method in a specification, our infrastructure uses the method name and the full name of the method's declaring class as the query of GCSE to search for related code snippets.

To parse partial code of downloaded code snippets, our infrastructure uses a partial parser (Dagenais and Hendren 2008) to resolve class types and to build control-flow graphs. Our infrastructure uses inter-procedural analysis but limits the analysis within the same class since code snippets from GCSE are partial. After that, our infrastructure checks whether the downloaded code snippets violate the inferred specification using the resolved types and the built control-flow graphs.

Our infrastructure uses the criteria as shown in Sect. 2.3 to detect violations of inferred specifications. In particular, if a resource is declared by a method as a local variable, our infrastructure checks the method's control-flow using the criteria defined in Sect. 2.3. If a resource is declared by a class as a field, our infrastructure checks whether two types of methods involved in the criteria are both called in the class's methods. For example, if a class declares a file as a field and opens the file in a method of the class, the class should also close the file in some method of the class. Otherwise, the class contains an "*Cr/Cl*" violation. Note that this requirement may be too strict and thus cause false positives.

6.3.2 Detected violations

Table 6 shows the results. Column "*API clients*" lists the number of client projects with violations. These API clients are all from released versions of mature software. Column "*Violations*" lists the number of code snippets with found violations. Its subcolumns list the number of violations detected by the corresponding criterion.

We manually inspected violations detected by our infrastructure. Among the violations, we identified those that we were able to determine not to be defects. We refer to these identified violations as false positives. Furthermore, we investigated the possible reasons that cause the false positives. We refer to the remaining violations as suspected defects.

Due to human factors for determining defects, these suspected defects may contain both false and real defects. We further analyze the causes of these suspected

---

[12]http://www.google.coM/Clodesearch.

Table 7 Results of violation analysis

Library	Suspected defects				False positives			
	Cr/Cl	*L/U*	*M/Cl*	*Cr/M*	*Spec*	*Partial*	*Strict*	*Doc*
J2SE	2	0	9	0	4	17	16	34
J2EE	2	0	14	0	2	8	6	10
JBoss	4	11	31	0	7	59	34	44
iText	4	0	0	0	0	4	2	5
Oracle	11	0	12	0	0	13	11	7
Total	100				283			

defects. Table 7 shows the results. Column "*Suspected defects*" lists the number of code snippets with suspected defects. Sub-columns of column "*Suspected defects*" list the number of suspected defects detected by the corresponding criterion. Column "*False positives*" lists the number of false positives, and its sub-columns list the numbers of false positives caused by different factors. In particular, sub-column "*Spec*" represents false positives caused by incorrectly inferred specifications. Sub-column "*Partial*" represents false positives caused by the imprecision of partial analysis. Sub-column "*Strict*" represents false positives caused by the strict requirement in our infrastructure to detect defects. For example, it is possible that a class returns the file to other classes and lets other classes close the file, and such a situation causes false positives. Sub-column "*Doc*" represents false positives caused by flaws in API documentation (see Sect. 8.3 for such an example).

From the results in Tables 6 and 7, we have the following observations. First, our infrastructure detected 383 violations in total, including 283 false positives. That is to say, 73.9% of the found violations are false positives (see Sect. 8.2 for the discussion on the false positive rate). Second, most of the found violations are "*Cr/Cl*" and "*M/Cl*" violations, and we did not find any "*Cr/M*" violation. We suspect the reason to be that most code snippets from open source projects may have been tested by developers. As "*Cr/M*" violations can cause serious problems such as exceptions that are easy to observe, developers may have found these violations and fixed them, whereas other violations may cause problems such as memory leaks that are not easy to observe. Third, as shown in sub-column "*Spec*" of Table 6, incorrectly inferred specifications are not the main factor of false positives. Finally, many false positives are caused by flaws in API documentation. This factor seems to reflect a disadvantage of our approach. However, as these false positives can draw library developers' attention to flaws in API documentation, these library developers may use the reported violations to improve the quality of API documentation (see Sect. 7 for details).

6.3.3 Confirmed defects

To better validate the suspected defects, we used the following procedure to determine whether they are real defects. First, we checked the latest version of the project to determine whether a suspected defect is already fixed. If so, we deemed that we found a previously known real defect. For example, with our infrastructure, we found a suspected defect of an unclosed input stream in the JBoss application server as shown

<div align="center">WebServer.java (version 2.4.11)</div>

```
protected byte[] getBytes(URL url) throws IOException {
  InputStream in = new BufferedInputStream(url.openStream());
  log.debug("Retrieving "+url.toString());
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  byte[] tmp = new byte[1024];
  int bytes;
  while ((bytes = in.read(tmp)) != -1) {
    out.write(tmp, 0, bytes);
  }
  return out.toByteArray();
}
```

<div align="center">WebServer.java (version 4.2.0)</div>

```
protected byte[] getBytes(URL url) throws IOException {
  InputStream in = new BufferedInputStream(url.openStream());
  if (log.isDebugEnabled())  log.debug("Retrieving " + url);
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  byte[] tmp = new byte[1024];
  int bytes;
  while ((bytes = in.read(tmp)) != -1) {
    out.write(tmp, 0, bytes);
  }
  in.close();
  return out.toByteArray();
}
```

Fig. 13 A confirmed defect in the JBoss application server

Fig. 14 The partial inferred specification for `java.io.InputStream`

in the left code snippet of Fig. 13. Our infrastructure detected suspected defect since it violates an inferred specification. In the J2SE's Javadoc, the method description of the `read(byte[] b)` method is "*Reads some number of bytes from the input stream and stores them into the buffer array b*", and the method description of the `close()` method is "*Closes this input stream and releases any system resources associated with the stream*". Figure 14 shows the corresponding inferred specification. The suspected defect violates the specification since it never closes an input stream after it reads its contents.

We checked JBoss's latest version and confirmed that this suspected defect is a real defect. In particular, the left code snippet of Fig. 13 shows the found suspected defect in JBoss 2.4.11, and the right code snippet of Fig. 13 shows how the suspected defect is fixed in JBoss 4.2.0. Second, if a suspected defect of a project is not fixed even in the latest version, we submitted the suspected defect as a defect report to the project's defect repository or contacted the project's developers through emails. If developers of the project confirmed that the suspected defect is a real defect, we deemed that we found a previously unknown defect. If developers of the project confirmed that a suspected defect is not a defect, we deemed it as a false defect. For those bug reports or bug-reporting emails that developers of the project have not responded yet, we deemed them as pending defects.

Table 8 shows the results. Column "*Checked*" lists total numbers of checked defects by the preceding procedure. Column "*Confirmed*" lists numbers of confirmed defects, and its sub-columns list the numbers of previously known and unknown defects. Column "*False*" lists numbers of suspected defects that the developers confirmed as false defects. The results of Table 8 show that we found 35 confirmed real defects through the preceding procedure, including 5 previously unknown defects.

```
protected DocData getNextDocData() throws ...{
  ...
  BufferedReader reader = new BufferedReader( new FileReader(f));
  String dateStr = reader.readLine();
  ...
  return dd;
}
```

Table 8 Confirmed defects by developers

Library	Checked	Confirmed		False
		Known	*Unknown*	
J2SE	10	9	0	1
J2EE	8	7	0	1
JBoss	19	14	1	4
iText	0	0	0	0
Oracle	4	0	4	0
Total	41	30	5	6

Table 9 Confirmed defects by testing

Library	Tested	Confirmed	False
J2SE	1	1	0
J2EE	8	7	1
JBoss	27	25	2
iText	4	1	3
Oracle	19	19	0
Total	59	53	6

We further manually construct test cases to confirm the rest of suspected defects, and Table 9 shows the results. Column "*Tested*" lists the number of suspected defects under testing. Column "*Confirmed*" lists the number of confirmed defects. Column "*False*" lists the number of suspected defects that we cannot construct appropriate test cases to reveal these defects. The results of Table 9 show that we found 53 confirmed defects through testing. The appendix of our paper lists some found defects.

In summary, from the results of Tables 8 and 9, we find that 88 confirmed defects totally. Among them, developers of the open source projects confirmed 5 defects, which were previously unknown. The results confirm that inferred specifications are useful to detect defects in true practice.

6.4 Threats to validity

The threat to external validity includes the representativeness of the subjects in true practice. Although we applied our approach on the Javadocs of five open source and commercial libraries, our approach is evaluated only on the Javadocs of these limited libraries. The threat could be reduced by more evaluations on more subjects in future work. The threat to internal validity includes human factors for determining defects. To reduce the threat, we inspected defects carefully and contacted developers to confirm these defects. The threats could be further reduced by involving more experienced developers in future evaluations. The threat to internal validity also includes human factors for tagging actions and resources to train our model. To reduce the threat, we tagged these actions and resources carefully. The threats could be further reduced by involving more experienced developers and tagging more documents for our training process.

7 Benefits of our approach

To our knowledge, our work is the first approach that infers specifications from API documentation. In this section, we analyze the benefits of our approach over existing approaches or practices.

7.1 Mining specifications from client code

Our previous work (Thummalapenta and Xie 2008) shows that coldspots are quite common in libraries. Coldspots of libraries represent those methods and classes that are rarely used by existing client code. In particular, our previous studies (Thummalapenta and Xie 2008) show that in all eight widely used libraries, coldspots are more than the sums of hotspots and neutrals. In literature, most existing approaches mine specifications from client code (see Sect. 4.1 for details), and these approaches may fail to mine specifications for those coldspots since coldspots have no sufficient client code. Although coldspots are not as popular as those hotspots in some cases, programmers may still use coldspots, and can introduce defects into code under development if they do not use coldspots correctly. As our approach does not need client code, it is able to infer specifications for both hotspots and coldspots, complementing existing approaches.

7.2 Inferring specifications from comments

Tan et al. (2007) proposed the first approach that infers rules from comments within source code. We used all the rule templates listed in Table 2 of their paper (Tan et al. 2007) to query the API documentation of J2SE 1.5, and we found only 11 exactly matched sentences. Padioleau et al. (2009) report that iComment (Tan et al. 2007) leveraged only 1% of comments since most comments do not explicitly contain rules. Our approach is able to infer various specifications from API documentation, complementing their approaches.

7.3 Detecting defects in API documentation

As shown in Table 6, many false positives are caused by defects in API documentation. For example, Fig. 10a shows an inferred specification for the resource socket@java.net.Socket. When we used this specification to find defects, we found that in many code snippets, close() is not called after connect(). We further checked these code snippets, and we found that in the J2SE library, a socket is often associated with an input stream. When the input stream is closed, the socket is automatically closed. As this usage is contrary to normal expectations, some developer has submitted a defect report to the J2SE defect database[13] (see *Defect #4118429* for details). This reported defect is confirmed as a real defect by J2SE developers. Although we count these violations as false positives in Table 6, our observation suggests that our approach can also be used to find defects in API documentation.

---

[13]http://defects.sun.com/defectdatabase/.

In addition, some inferred specifications are imbalanced, and these imbalanced specifications can provide insights for library developers to improve API implementation or documentation. For example, the resource shown in Fig. 14 has a closure method, but does not have a corresponding creation method. We checked its document[14], and we found that the resource has no explicit creation methods. Library developers may implement such a method, or add documents to explain the existing implicit creation method (*i.e.*, the `InputStream()` constructor).

8 Discussion and future work

In this section, we discuss various issues that are related to our approach.

8.1 Resource analysis

One class/interface may have more than one resource. For each resource, our approach infers an automaton. For example, Figs. 10c and d show two inferred specifications for one class `WebModuleFactory`. If we simply select those common concrete nouns as resources, we would mix the two automata and infer a false specification because from their documentation[15] the descriptions of the four methods use the same common concrete noun "*JSR-77*". Resource analysis may help us infer more complicated specifications. In the preceding example, if we consider the relation between `WebModule` and `Servlet`, it may be feasible to combine the specification shown in Fig. 10c and the specification shown in Fig. 10d into a more complicated specification. In future work, we plan to take semantic relationships among resources into consideration and infer more complicated specifications.

In addition, we found that some resources of different names refer to the same resource. For example, "*Document*", "*RtfDocument*", and "*RTF document*" refer to the same resource in the document for `com.lowagie.text.rtf.RtfWriter2`. The current implementation of Doc2Spec cannot group methods of the three resources into one category yet. In NLP, the problem of resolving noun phrases to one real-world entity is known as coreference resolution (Hirschman 1997). We plan to leverage these techniques to infer better specifications in future work.

8.2 False positive rate

As shown in Table 6, the false positive rate of our approach is 73%. It is reasonable due to four factors. First, some found defects in documentation are interpreted as "false positive" instead of "true positive". Second, we use all inferred specifications for detecting defects instead of using some selected specifications as some previous work (*e.g.*, Yang et al. 2006) did. Third, static checkers typically produce high false positive rate (*e.g.*, 76% reported by Williams and Hollingsworth 2005). Finally, due to

---

[14]http://download.oracle.com/javase/1.5.0/docs/api/java/io/InputStream.html.

[15]http://docs.jboss.org/jbossas/javadoc/4.0.5/management/org/jboss/management/j2ee/factory/WebModuleFactory.html.

the intrinsic difficulty in parsing partial code, the resolved types and the built control-flow graphs are not fully accurate. As a result, our approach may cause more false positives than traditional static checkers.

Even under the negative impact of the preceding factors, our false positive rate is comparable with other approaches (*e.g.*, 63% reported by Tan et al. 2007). Indeed, reducing false positives is quite important, and we plan to reduce our false positive rate in our future work. For example, some descriptions contain words such as "*has to be closed*". If the description of a method has such words, we can have more confidence that the method belongs to a specific type of methods (*e.g.*, closure methods in this example). If our approach takes these words into consideration, we may further reduce false positives of our approach. As another example, for each method description, our approach chooses an action and a resource both with the highest scores to build one action-resource pair. If we keep multiple action-resource pairs for each method description and apply client-code analysis to choose the best pair, we may increase accuracies of inferred specifications, and thus can reduce false positives.

8.3 Extensions of our approach

8.3.1 Defects in local code bases

In this paper, we developed an infrastructure to check code snippets returned from GCSE. The infrastructure helps us detect various violations to show the usefulness of inferred specifications. To help developers find defects using inferred specifications, we plan to adapt our defect-detection infrastructure also for local code bases in future work. As our evaluations confirm that our approach infers various useful specifications to detect real defects, we expect the adapted infrastructure to be useful for developers to detect defects in local code bases. Our adapted infrastructure could produce fewer false positives as it does not have to rely on partial analysis for local code bases whose source files are often complete. Although our tool finds only a limited number of defects in our evaluation, we expect our extended tool to find more defects in local code bases. As our approach can help detect and remove defects related to violating important resource usages, it should be worth taking the effort to apply our approach in practice.

8.3.2 Mining specification templates

Our approach relies on a predefined specification template for inferring resource specifications. In practice, some resource usages may be quite complicated and cannot be instantiated with our predefined template. As discussed in Sect. 4, there are many approaches that mine rich specifications from various data. We plan to mine specification templates from existing specifications mined by those approaches. After mining specification templates, we can further improve our approach to mine more complicated resource usages.

8.3.3 Other API documentation and descriptions

API documentation other than Javadocs may follow different conventions to describe actions and resources. In addition, descriptions of parameters, return values, and exception throws may also contain useful information to infer specifications. We need to tag specific training corpora for other API documentation that follows quite different conventions. We also need to extend our HMM model to deal with other descriptions and explore whether these descriptions help our specification inference in future work.

8.3.4 Analyzing library code

Library code analysis may help infer specifications for methods without descriptions. For example, Fry et al. (2008) proposed an approach that can extract verb-direct object pairs from method signatures. We plan to adapt their approach to extract action-resource pairs from method signatures for methods without descriptions in future work. As another example, Buse and Weimer (2008) proposed an approach to generate comments for *exception* clauses via code analysis. We plan to adapt their approach to generate descriptions for methods without descriptions.

Library code analysis may also be useful for methods with descriptions. For example, Høst and Østvold (2009) proposed an approach that detects inconsistencies between method names and method code. Our previous work (Zhong et al. 2008b) can infer specifications from library code statically. In future work, we plan to investigate inconsistencies between specifications inferred from code with specifications inferred from documentation.

8.3.5 Writing resource specifications

Due to the heavy effort to write specifications, library developers often do not provide written specifications. Some researchers have proposed approaches to reduce the effort of writing specifications. For example, Henkel and Diwan (2004) proposed an approach that is able to help write algebraic specifications. With these approaches, library developers may consider to write resource specifications manually, and our approach can further improve these approaches. For example, an approach can reduce the effort of writing resource specifications, if the approach leverages Doc2Spec to infer candidate specifications, and allows library developers to revise those false ones.

9 Conclusion

Although many correct API usages are already written in API documentation, developers may still produce defects related to resources even when correct usages of these resources are already described in API documentation. In this paper, we propose a novel approach that infers resource specifications from existing API documentation. We conducted evaluations on the Javadocs of five widely used libraries. The results

show that our approach infers various specifications with relatively high precisions, recalls, and F-scores. We further use inferred specifications to detect defects. The results show that resource specifications inferred by our approach are useful to detect real defects in practice.

Appendix

Some example confirmed defects are as follows[16].

Unclosed resources in normal code. We find that normal client code may have defects since some resources are never closed. These defects belong to the second type and the fourth type of defects listed in Sect. 2.3. For example, we found a previously known defect in a code snippet of the *Apache Lucene* project[17] since its developer does not close a resource referred by a local variable named reader.

Another previously known defect was found in the code snippet of the *Spring Framework* project[18] since its developer does not close a resource referred by a local variable named producer.

```
protected void doSend(Session session, Destination destination,
                                       MessageCreator messageCreator) throws ...{
  MessageProducer producer = createProducer(session, destination);
  ...
  doSend(producer, message);
  ...
}

protected void doSend(MessageProducer producer, Message message) throws ...{
  ...
  producer.send(message);
  ...
}
```

We found a defect in the code snippet of the *Globus* project[19] by testing since its developer does not close a resource referred by a local variable named ix.

```
public String ejbCreate(...) throws ...{
  try{
     Interaction ix = this.jmCon.createInteraction();
     ...
```

---

[16]Our infrastructure found these defects in August 2008. As the GCSE's repository evolves quickly, many URLs of buggy snippets have now become invalid, so we do not provide these URLs from GCSE here.

[17]http://lucene.apache.org.

[18]http://www.springframework.org.

[19]http://www.globus.org.

```
    Record oRec = ix.execute(iSpec, iRec);
    Iterator iterator = ((IndexedRecord)oRec).iterator();
    this.primaryKey = (String)iterator.next();
    return this.primaryKey;
  } catch (ResourceException rex) {
    throw new EJBException("ejbCreate: " + ...);
  }
}
```

We suspected that the developer of the code snippet forgets to close the resource referred by the variable since we found that in the same code snippet, the developer does close a resource referred by another variable also named as `ix` in a method as follows.

```
public void cancel() throws ...{
  try{
    Interaction ix = this.jmCon.createInteraction();
    ...
    Record oRec = ix.execute(iSpec, iRec);
    ...
    ix.close();
    return;
  } catch (ResourceException rex) {
    throw new EJBException("cancel(): " + ...);
  }
}
```

Unclosed resources in exception handling. We find that exception handling code may have defects since some resources are never closed when exceptions are thrown. These defects belong to the second type and the fourth type of defects listed in Sect. 2.3. For example, we found a previously unknown defect in a code snippet of the project *TopX*[20] since its developer does not close `rRset` in its *exception* block. This defect is not fixed even in the latest version and is confirmed by the developers of *TopX* through emails.

```
public double getMinimumScore(){
  ...
  try{
    rRset = mStmt.executeQuery("...");
    ...
    rRset.close();
  } catch (SQLException e) {
    e.printStackTrace();
  }
}
```

We found a defect by testing in a code snippet of the *SIM-PL* project[21] since its developer does not close `document` in its *exception* block either.

```
public double save(...) throws ...{
  ...
  try{
    ...
    com.lowagie.text.Document document = new com.lowagie.text.Document(
                new com.lowagie.text.Rectangle(bb.width, bb.height));
    ...
    document.open();
    ...
    document.close();
```

---

[20]http://topx.sourceforge.net.

[21]http://www.science.uva.nl/amstel/SIM-PL/.

```
    } catch (DocumentException ex) {
      throw new IOException(ex.toString());
    }
}
```

Defects in tutorials. We even found some resource-relevant defects in code snippets of tutorials. We suspect that these resources and their related methods are relatively unfamiliar to developers and thus it is difficult for even experienced developers who write tutorials. For example, we found a defect in the code snippet from an Oracle's tutorial[22] as its developer does not free a temporary CLOB object named `clob`.

```
private void processDocument() throw Exception{
  try{
    ...
    CLOB clob = CLOB.createTemporary(conn, true, CLOB.DURATION_SESSION );
    ...
  } catch(NullPointerException ex ){
    Alert.log(...);
  }
}
```

However, in the code snippet from another Oracle's tutorial[23] as follows, its developer does free a created temporary CLOB object.

```
private void doBulkLoad(...) throw Exception{
  try{
    ...
    m_TemporaryCLOB = CLOB.createTemporary( connection, true, CLOB.DURATION_SESSION );
    ...
    m_TemporaryCLOB.freeTemporary(  );
  } catch(NullPointerException ex ){
    ...
  }
}
```

In summary, using specifications inferred by Doc2Spec, we found various previously known and unknown defects that are related to resource usages from open source projects. The results demonstrate the usefulness of our inferred specifications to detect defects. Based on our results, developers did produce source code that is inconsistent with the resource usages described in API documentation, and these inconsistencies can indicate defects. Our inferred specifications are useful to detect defects since these specifications help detect the inconsistencies between API documentation and source code.

References

Acharya, M., Xie, T.: Mining API error-handling specifications from source code. In: Proc. Fundamental Approaches to Software Engineering, pp. 370–384 (2009)

Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: From usage scenarios to specifications. In: Proc. 6th ESEC/FSE, pp. 25–34 (2007)

Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: Proc. 32nd POPL, pp. 98–109 (2005)

---

[22] http://www.oracle.com/technology/sample_code/tech/xml/xmldb/saxloader/oracle/otnsamples/xmldb/saxloader/examples/DocumentReader.java.html.

[23] http://www.oracle.com/technology/sample_code/tech/xml/xmldb/simplebulkloader/oracle/otnsamples/xmldb/simplebulkloader/examples/simplebulkloader.java.html.

Ambriola, V., Gervasi, V.: Processing natural language requirements. In: Proc. 12th ASE, pp. 36–45. IEEE Computer Society, Los Alamitos (1997)

Ammons, G., Bodík, R., Larus, J.: Mining specifications. In: Proc. 29th POPL, pp. 4–16 (2002)

Anvik, J., Hiew, L., Murphy, G.: Who should fix this bug? In: Proc. 28th ICSE, pp. 361–370 (2006)

Arnout, K., Meyer, B.: Uncovering hidden contracts: The .NET example. Computer **36**(11), 48–55 (2003)

Baum, L., Petrie, T., Soules, G., Weiss, N.: A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. Ann. Math. Stat. 164–171 (1970)

Buse, R., Weimer, W.: Automatic documentation inference for exceptions. In: Proc. ISSTA, pp. 273–282 (2008)

Buse, R., Weimer, W.: Automatically documenting program changes. In: Proc. 26th ASE, pp. 33–42 (2010)

Chinchor, N.: MUC-7 named entity task definition. In: Proc. 7th MUC (1997)

Cohen, W., Sarawagi, S.: Exploiting dictionaries in named entity extraction: combining semi-Markov extraction processes and data integration methods. In: Proc. 10th KDD, pp. 89–98 (2004)

Dag, J., Regnell, B., Gervasi, V., Brinkkemper, S.: A linguistic-engineering approach to large-scale requirements management. IEEE Softw. **3**, 3 (2005)

Dagenais, B., Hendren, L.J.: Enabling static analysis for partial Java programs. In: Proc. 23rd OOPSLA, pp. 313–328 (2008)

Dekel, U., Herbsleb, J.D.: Reading the documentation of invoked API functions in program comprehension. In: Proc. 17th ICPC, pp. 168–177 (2009a)

Dekel, U., Herbsleb, J.D.: Improving API documentation usability with knowledge pushing. In: Proc. 31st ICSE, pp. 320–330 (2009b)

Engler, D., Chen, D., Chou, A.: Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In: Proc. 18th SOSP, pp. 57–72 (2001)

Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Applications of linguistic techniques for use case analysis. Requir. Eng. **8**(3), 161–170 (2003)

Fellbaum, C., et al.: WordNet: An Electronic Lexical Database. MIT Press, Cambridge (1998)

Fry, Z., Shepherd, D., Hill, E., Pollock, L., Vijay-Shanker, K.: Analysing source code: looking for useful verb-direct object pairs in all the right places. IET Softw. **2**(1), 27–36 (2008)

Gabel, M., Su, Z.: Symbolic mining of temporal specifications. In: Proc. 13th ICSE, pp. 51–60 (2008)

Gabel, M., Su, Z.: Online inference and enforcement of temporal properties. In: Proc. 32nd ICSE, pp. 15–24 (2010)

Gegick, M., Rotella, P., Xie, T.: Identifying security bug reports via text mining: An industrial case study. In: Proc. 7th MSR, pp. 11–20 (2010)

Gervasi, V., Zowghi, D.: Reasoning about inconsistencies in natural language requirements. ACM Trans. Softw. Eng. Methodol. **14**(3), 277–330 (2005)

Goldin, L., Berry, D.: AbstFinder, a prototype natural language text abstraction finder for use in requirements elicitation. Autom. Softw. Eng. **4**(4), 375–412 (1997)

Gowri, M., Grothoff, C., Chandra, S.: Deriving object typestates in the presence of inter-object references. In: Proc. 20th OOPSLA, pp. 77–96 (2005)

Hayes, J., Dekhtyar, A., Sundaram, S.: Advancing candidate link generation for requirements tracing: The study of methods. IEEE Trans. Softw. Eng. **32**(1), 4–19 (2006)

Henkel, J., Diwan, A.: A tool for writing and debugging algebraic specifications. In: Proc. 26th ICSE, pp. 449–458 (2004)

Hirschman, L.: MUC-7 coreference task definition. In: Proc. 7th MUC (1997)

Horie, M., Chiba, S.: Tool support for crosscutting concerns of API documentation. In: Proc. 8th AOSD, pp. 97–108 (2010)

Høst, E.W., Østvold, B.M.: Debugging method names. In: Proc. 23rd ECOOP, pp. 294–317 (2009)

Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Trans. Program. Lang. Syst. **27**(2), 264–313 (2005)

Jeong, G., Kim, S., Zimmermann, T.: Improving bug triage with bug tossing graphs. In: Proc. 7th ESEC/FSE, pp. 111–120. ACM, New York (2009)

Kof, L.: Scenarios: Identifying missing objects and actions by means of computational linguistics. In: Proc. 15th RE, pp. 121–130 (2007)

Kremenek, T., Twohey, P., Back, G., Ng, A., Engler, D.: From uncertainty to belief: Inferring the specification within. In: Proc. 7th OSDI, pp. 259–272 (2006)

Lee, C., Chen, F., Rosu, G.: Mining parametric specifications. In: Proc. 33rd ICSE, pp. 591–600 (2011)

Li, Z., Zhou, Y.: PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In: Proc. ESEC/FSE, pp. 306–315 (2005)

Livshits, V., Zimmermann, T.: Dynamine: Finding common error patterns by mining software revision histories. In: Proc. ESEC/FSE, pp. 31–40 (2005)

Lo, D., Khoo, S.: SMArTIC: towards building an accurate, robust and scalable specification miner. In: Proc. 14th FSE, pp. 265–275 (2006)

Lo, D., Maoz, S.: Scenario-based and value-based specification mining: better together. In: Proc. 25th ASE, pp. 387–396 (2010)

Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In: Proc. 13th ASPLOS, pp. 329–339 (2008)

Meziane, F., Athanasakis, N., Ananiadou, S.: Generating natural language specifications from UML class diagrams. Requir. Eng. **13**(1), 1–18 (2008)

Mikheev, A., Moens, M., Grover, C.: Named entity recognition without gazetteers. In: Proc. 9th EACL, pp. 1–8 (1999)

Novick, D., Ward, K.: Why don't people read the manual. In: Proc. 24th SIGDOC, pp. 11–18 (2006)

Olson, D.: Advanced Data Mining Techniques. Springer, Berlin (2008)

Padioleau, Y., Tan, L., Zhou, Y.: Listening to programmers—Taxonomies and characteristics of comments in operating system code. In: Proc. 31st ICSE, pp. 331–341 (2009)

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

Raman, A., Patrick, J.: The sk-strings method for inferring PFSA. In: Proc. Machine Learning Workshop Automata Induction, Grammatical Inference, and Language Acquisition (1997)

Ramanathan, M., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: Proc. 29th ICSE, pp. 240–250 (2007)

Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. In: Machine Learning: From Theory to Applications, pp. 51–73 (1993)

Robillard, M.P., DeLine, R.: A field study of API learning obstacles. Empir. Softw. Eng. (2011). doi:10.1007/s10664-010-9150-8

Runeson, P., Alexandersson, M., Nyholm, O.: Detection of duplicate defect reports using natural language processing. In: Proc. 29th ICSE, pp. 499–510 (2007)

Sawyer, P., Rayson, P., Garside, R.: REVERE: Support for requirements synthesis from documents. Inf. Syst. Front. **4**(3), 343–353 (2002)

Shepherd, D., Fry, Z., Hill, E., Pollock, L., Vijay-Shanker, K.: Using natural language program analysis to locate and understand action-oriented concerns. In: Proc. 6th AOSD, pp. 212–224 (2007)

Shi, L., Zhong, H., Xie, T., Li, M.: An empirical study on evolution of API documentation. In: Proc. FASE, pp. 416–431 (2011)

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L.L., Vijay-Shanker, K.: Towards automatically generating summary comments for Java methods. In: Proc. 25th ASE, pp. 43–52 (2010)

Stylos, J., Faulring, A., Yang, Z., Myers, B.: Improving API documentation using API usage information. In: Proc. IVL/HCC, pp. 119–126 (2009)

Tan, L., Yuan, D., Krishna, G., Zhou, Y.: /* iComment: Bugs or Bad Comments?*/. In: Proc. 21st SOSP, pp. 145–158 (2007)

Thummalapenta, S., Xie, T.: SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In: Proc. 23rd ASE, pp. 327–336 (2008)

Thummalapenta, S., Xie, T.: Mining exception-handling rules as sequence association rules. In: Proc. 31th International Conference on Software Engineering, May 2009, pp. 496–506 (2009a)

Thummalapenta, S., Xie, T.: Alattin: Mining alternative patterns for detecting neglected conditions. In: Proc. 24th Automated Software Engineering, pp. 283–294 (2009b)

Viterbi, A.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans. Inf. Theory **13**(2), 260–269 (1967)

Wang, X., Zhang, L., Xie, T., Anvik, J., Sun, J.: An approach to detecting duplicate bug reports using natural language and execution information. In: Proc. 30th ICSE, pp. 461–470 (2008)

Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: Proc. ESEC/FSE, pp. 35–44 (2007)

Weimer, W., Necula, G.: Mining temporal specifications for error detection. In: Proc. TACAS, pp. 461–476 (2005)

Whaley, J., Martin, M., Lam, M.: Automatic extraction of object-oriented component interfaces. In: Proc. ISSTA, pp. 218–228 (2002)

Williams, C., Hollingsworth, J.: Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans. Softw. Eng. **31**(6), 466–480 (2005)

Würsch, M., Ghezzi, G., Reif, G., Gall, H.: Supporting developers with natural language queries. In: Proc. 32nd ICSE, pp. 165–174 (2010)

Xu, G., Rountev, A.: Precise memory leak detection for Java software using container profiling. In: Proc. 30th ICSE, pp. 151–160 (2008)

Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proc. 28th ICSE, pp. 282–291 (2006)

Zhong, H., Zhang, L., Mei, H.: Early filtering of polluting method calls for mining temporal specifications. In: Proc. 15th APSEC, pp. 9–16 (2008a)

Zhong, H., Zhang, L., Mei, H.: Inferring specifications of object oriented APIs from API source code. In: Proc. 15th APSEC, pp. 221–228 (2008b)

Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: Mining and recommending API usage patterns. In: Proc. 23rd ECOOP, pp. 318–343 (2009a)

Zhong, H., Zhang, L., Xie, T., Mei, H.: Inferring resource specifications from natural language API documentation. In: Proc. 24th ASE, pp. 307–318 (2009b)

Zhou, G., Su, J.: Named entity recognition using an HMM-based chunk tagger. In: Proc. 40th ACL, pp. 473–480 (2001)