Inferring Resource Specifications from Natural Language API Documentation

Hao Zhong[1,2], Lu Zhang[1,2,*], Tao Xie[3,*], Hong Mei[1,2,*]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China
[2]Institute of Software, School of Electronics Engineering and Computer Science, Peking University, China
[3]Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA
{zhonghao04, zhanglu}@sei.pku.edu.cn, xie@csc.ncsu.edu, meih@sei.pku.edu.cn

***Abstract*—Typically, software libraries provide API documentation, through which developers can learn how to use libraries correctly. However, developers may still write code inconsistent with API documentation and thus introduce bugs, as existing research shows that many developers are reluctant to carefully read API documentation. To find those bugs, researchers have proposed various detection approaches based on known specifications. To mine specifications, many approaches have been proposed, and most of them rely on existing client code. Consequently, these mining approaches would fail to mine specifications when client code is not available. In this paper, we propose an approach, called Doc2Spec, that infers resource specifications from API documentation. For our approach, we implemented a tool and conducted an evaluation on Javadocs of five libraries. The results show that our approach infers various specifications with relatively high precisions, recalls, and F-scores. We further evaluated the usefulness of inferred specifications through detecting bugs in open source projects. The results show that specifications inferred by Doc2Spec are useful to detect real bugs in existing projects.**

I. Introduction

Nowadays, it is a common practice for software libraries to provide developers with Application Programming Interface (API) documentation through online access. One example of such API documentation is J2EE's Javadoc[1]. Developers can find much useful information such as class/interface hierarchies and method descriptions from API documentation and learn how to correctly use libraries.

Still, developers may write code that is inconsistent with API documentation and produce bugs, as existing research on developers' behavior [29] shows that developers tend to ignore information in API documentation. Among these bugs, many are related to resource usages [27]. To detect resource bugs, researchers [21] propose various approaches that analyze resource usages, and these approaches need specifications that describe correct usages.

As API documentation contains much information on resource usages, it is feasible to infer resource specifications from API documentation. For example, the description of `java.sql.ResultSet.deleteRow()` is "*Deletes the current row from this ResultSet object and from the underlying database*", whereas 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 rule, an experienced developer can extract an implicit specification `deleteRow()` → `close()` 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.

However, it is challenging to infer resource specifications 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 Section III 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 infers various specifications with relatively high precisions, recalls, and F-scores.
- We further conducted an evaluation to detect bugs using inferred specifications. The results show that these specifications are useful to detect previously known or unknown bugs in open source projects.

The remainder of our paper is as follows. Section II illustrates our approach using an example. Section III presents related work. Section IV presents the details of our approach. Section V presents our evaluations. SectionVI presents benefits of our approach. Section VII discusses issues of our

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

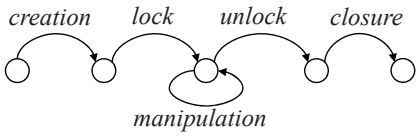

Figure 1. Specification template

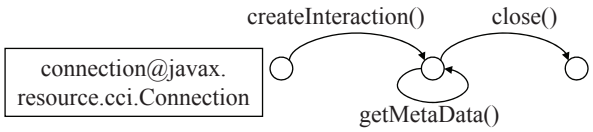

Figure 2. Resource specifications

approach. Section VIII concludes.

II. 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 bugs.

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 J2EE's Javadoc, our approach extracts three method descriptions of interface `javax.resource.cci.Connection` as follows:

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

`getMetaData()`:"*Gets the information on the underlying EIS 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()`:$\langle create, connection\rangle$.

`getMetaData()`:$\langle get, connection\rangle$.

`close()`:$\langle close, connection\rangle$.

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 Section VII-A for details). Our approach leverages an NLP technique to extract action-resource pairs accurately.

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()`→creation method.

`getMetaData()`→manipulation method.

`close()`→closure method.

Finally, in each category, our approach builds an automaton based on our predefined specification template shown in Figure 1. Figure 2 shows the inferred specification for the resource of interest. Our approach tailors our specification template to build the automaton shown in Figure 2 (see the end of Section IV-C for details).

Detecting bugs. To confirm the usefulness of an inferred specification, we need to investigate whether we can detect bugs 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 shown in Figure 2. 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 bug since unclosed connections may cause memory leaks. Such a case shows that the specification shown in Figure 2 is useful to detect bugs. 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 [32] warn "If you do not explicitly close your ResultSet and Statement objects, serious memory leaks could occur." The work proposed by Xu and Rountev [44] also focuses on memory leaks in Java programs.

III. Related Work

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

Generating 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.* [2] propose an approach and its supporting tool *Strauss* that uses an extended Angluin algorithm [33] to mine automata from execution traces that are related by traditional dataflow dependencies. Lo and Khoo [26] improve *Strauss* by introducing clustering techniques to refine traces before the mining process. Kremenek *et al.* [23] use Bayesian learning to match methods with a predefined automata template for specifications. Whaley *et al.* [42] mine automata-like models from execution traces and refine these models using code analysis. Gabel and Su [16] propose a symbolic algorithm based on binary decision diagrams to mine automata from execution traces. Alur *et al.* [1] use randomly generated test cases as clients and use Angluin's algorithm to infer automata from call sequences that do not throw exceptions. Gowri *et al.* [18] also use a client emulator as clients, and their approach uses some analysis results such as objects' relationships, internal states, and their specifications of libraries. Cook and Wolf [8] reduce the general problem of mining automata from execution traces to the classical grammar inference problem, and the problem has been proved to be NP-complete [17].

For sequence-based approaches, Engler *et al.* [12] use the Z-statistic value as the support to mine frequent call pairs from source files. Yang *et al.* [45] propose an algorithm to mine call pairs from execution traces. Weimer and Necula [41] propose an approach to mine and filter method pairs from execution traces. Li and Zhou [24] use frequent itemset mining to extract implicit programming properties and detect their violations for detecting bugs. Livshits and Zimmermann [25] propose an approach of mining properties from software revision histories. Wasylkowski *et al.* [40] use frequent sequence mining to mine frequent call sequences from clients and use anomalies to detect bugs in client code. Ramanathan *et al.* [34] use frequent sequence mining to mine frequent call sequences from execution traces extracted statically in client code. Zhong *et al.* [46] propose an approach that combines clustering with sequence mining to mine context-sensitive specifications from client code.

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 [38] 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 [3] propose an approach to manually extract contracts from .NET documents. Their inferred contracts consist of invariant-like preconditions and postconditions and do not capture legal call sequences as our approach does. Tan *et al.* [37] propose *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.

NLP in software engineering. As many software engineering activities involve natural languages, it is feasible to leverage NLP techniques to assist these activities. Kof [22] uses part-of-speech (POS) tagging to identify missing objects and actions in requirement documents. Sawyer *et al.* [35] use POS and semantic tagging to support requirement synthesis from documents. Fantechi *et al.* [13] use syntactic parsing to analyze uses cases from requirement documents. Shepherd *et al.* [36] use various NLP techniques such as stemming and POS tagging to locate and understand action-oriented concerns. 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.

Improving documents. Buse and Weimer [5] propose an approach to generate comments for *exception* clauses via code analysis. Dekel and Herbsleb [10], [11] propose eMoose that pushes and highlights those rule-containing sentences from API documentation for developers. Their approach improves the quality of documentation, whereas our approach infers specifications from API documentation and detects bugs in code.

IV. Approach

The overview of our approach is shown in Figure 3. Our approach focuses on API documentation in the form of Javadoc. Javadoc[2] is an industrial tool to generate API documentation for libraries in Java. In this paper, we also use "Javadoc" to denote API documentation generated by this tool. Figure 4 shows several pieces of J2SE's Javadoc. We next present detailed steps of our approach.

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

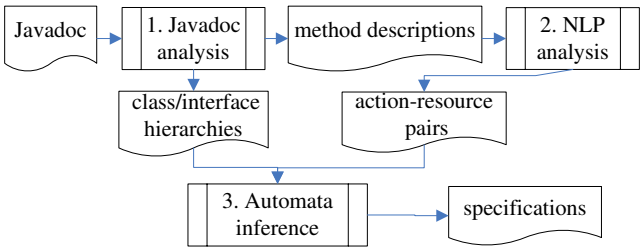

Figure 3. Overview of our approach

A. Javadoc analysis

The first step of our approach is to extract method descriptions and class/interface hierarchies from Javadocs. As shown in Figure 4, in Javadocs, method descriptions are under the topic "*Method Summary*", and class/interface hierarchies are under the topics "*Class Hierarchy*" and "*Interface Hierarchy*". 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 topic "*Method Summary*" by searching for the anchor name "*method_summary*". Some methods may have no descriptions, and our approach ignores these methods (see Section VII 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 topics "*Class Hierarchy*" and "*Interface Hierarchy*" through text matching.

B. NLP analysis

The second step of our approach is to build an action-resource pair from each method's description through NLP analysis. In NLP, the problem of identifying words belonging to a predefined category in a document is known as Named Entity Recognition (NER) [6]. 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 [28] 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 [7] 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[3] has a list of all players. Machine-learning-based approaches [48] 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 [48].

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

[3]http://mlb.com

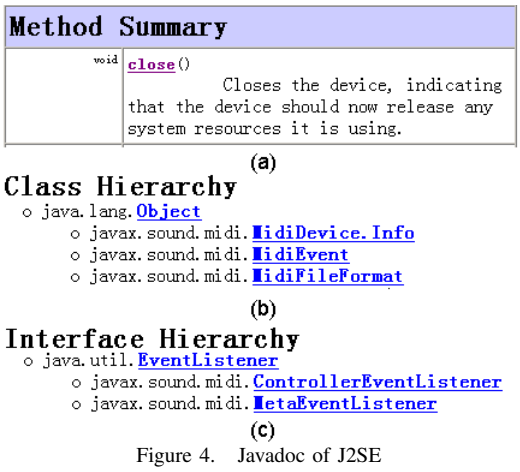

Figure 4. Javadoc of J2SE

- $\Omega_s = \{s_1, \ldots, s_n\}$ 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, $o_i = \langle w_i, f_i \rangle$ 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.
- $A : \Omega_s \times \Omega_s \rightarrow [0, 1]$ 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 [4] to train the parameters (A and B) from manually tagged method descriptions. The training process builds a model that describes characteristics of actions and resources. After training, our approach then uses the Viterbi algorithm [39] to tag method descriptions with scores based on the trained model.

For each method description, 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.

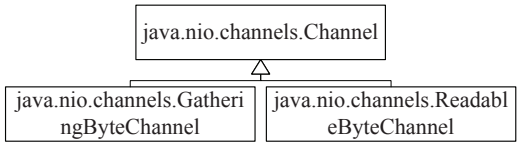

Figure 5. Hierarchical tree

C. 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 superclasses/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, Figure 5 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()`
$\langle close, channel\rangle$ ←"*Closes this channel.*"
`java.nio.channels.GatheringByteChannel.write()`
$\langle write, channel\rangle$←'*Writes a sequence of bytes to this channel from the given buffers.*"
`java.nio.channels.ReadableByteChannel.read()`
$\langle read, channel\rangle$←"*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. In our approach, we predefine five types of methods: creation, lock, manipulation, unlock, and closure:

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).

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*).

In each category, our approach maps its methods to the preceding types. 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 (*i.e.*, *WordNet* [14]). If using synonyms still fails to resolve a method's action, we map the method into a manipulation method since there can be various types of manipulations on a resource.

Our approach then builds an automaton for each category based on our predefined specification template (see Section VII-C 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 Section II, 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 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 Figure 2 from the specification template shown in Figure 1. 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 bugs.

V. Evaluations

We implemented a tool for our approach and conducted a series of evaluations using the tool. Our evaluations focus on two aspects of our approach: the performance of documentation analysis and specification inference (Section V-A) and the quality of inferred specifications (Section V-B).

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: J2SE[4], J2EE[5], JBoss[6], iText[7], and Oracle JDBC driver[8]. We conducted all the evaluations on a PC with an Intel Pentium 2.26 GHz CPU and 1512M memory running Windows 2000 professional.

[4]http://java.sun.com/javase/
[5]http://java.sun.com/javaee/
[6]http://www.JBoss.org
[7]http://www.lowagie.com/iText/
[8]http://tinyurl.com/6nj4x

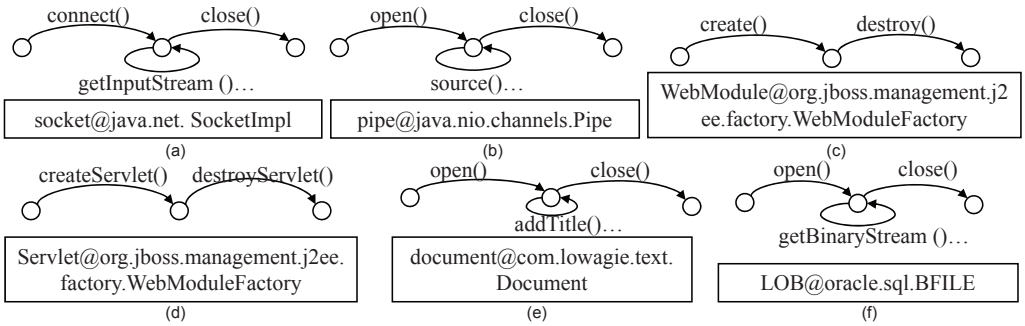

Figure 6. Example inferred specifications

Table I
PERFORMANCE OF DOC2SPEC

Lib	Version	#M	#D	#Spec	DT	ST
J2SE	5.0	25675	23829	3250	400.2	2357.2
J2EE	5.0	5670	5611	83	95.7	151.4
JBoss	4.0.5	26053	13869	373	430.4	140.0
iText	2.1.3	5846	4299	243	112.2	52.4
Oracle	10.1.0.5	2140	1916	32	28.3	22.4
Total		65384	49524	3981	1067.0	2723.4

A. Performance

To evaluate the performance of Doc2Spec, we recorded related data during documentation analysis and specification inference for each library, and Table I shows the results. Column "*Lib*" 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 "*#M*" lists the number of methods. Column "*#D*" lists the number of methods with descriptions. Column "*#Spec*" lists the number of inferred resource specifications. Column "*DT*" lists the time used to extract method descriptions and class/interface hierarchies in seconds. Column "*ST*" lists the time used to infer specifications based on the extracted information in seconds. Row "*Total*" lists total numbers for these columns.

From the results in Table I, 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. 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.

Another notable issue is 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 Section VII.

B. Quality of Inferred Specifications

Figure 6 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. To evaluate whether our approach infers various accurate specifications, we next present the statistics of the inferred specifications. As these inferred resource specifications are formal compared with documentation in natural languages, it is possible to use these specifications to detect bugs. To evaluate the usefulness of the inferred specifications, we next also present the results of using these specifications to detect bugs in open source projects.

1) Statistics of inferred resource specifications: Table I shows the basic statistics of the inferred resource specifications. To analyze the distribution of specifications in each library, we further present Figures 7 and 8. 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 Figure 7 shows that 69.1% of the specifications inferred from 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.

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 II shows the results. Column "*#H*" lists the number of selected hierarchies in each library. Column "*#S*" lists the number of manually built specifications for each selected hierarchy. In statistical classification [30], *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

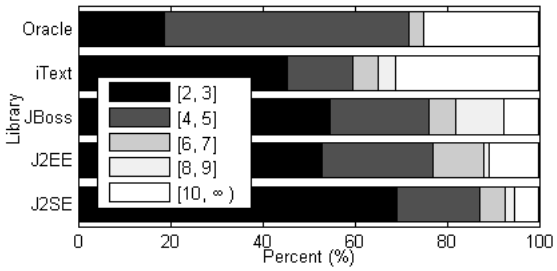

Figure 7. Percentages of specifications that involve specific numbers of methods

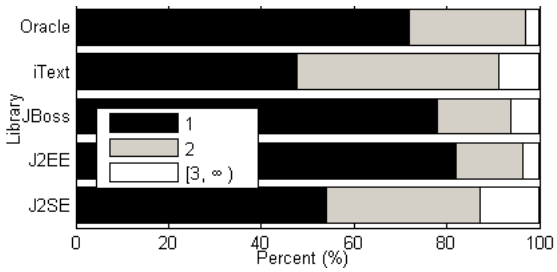

Figure 8. Percentages of specifications that involve specific numbers of classes/interfaces

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.

$$Precison = \frac{true\ positives}{true\ positives + false\ positives} \quad (1)$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \quad (2)$$

$$F-score = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (3)$$

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 bugs instead of a whole specification.

The results show that our approach achieves relatively high 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.

2) Usefulness of inferred resource specifications: We further used the specifications inferred by Doc2Spec to detect bugs in open source projects to evaluate the usefulness of these specifications.

We implemented an infrastructure to automate our evaluation. For a specification, our infrastructure first searches for the specification's related code snippets from the Internet using Google code search engine[9] (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 [9] 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. Finally, our infrastructure checks whether the downloaded code snippets violate the inferred specification using the resolved types and the built control-flow graphs. In particular, if a resource is declared by a method as a local variable, our infrastructure checks the method's control-flow graph using the following criteria:

- *O/M*: A manipulated resource should be already created if a resource has manipulation methods and creation methods.
- *O/C*: 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/C*: 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 *O/M* and *O/C* are not applicable.

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 "*O/C*" violation. Note that this requirement may be too strict and thus cause false positives.

[9]http://www.google.com/codesearch

Table III
RESULTS OF FOUND VIOLATIONS AND SUSPECTED BUGS AMONG THEM

Library	API clients	Violations				Suspected bugs				False positives			
		O/C	*L/U*	*M/C*	*O/M*	*O/C*	*L/U*	*M/C*	*O/M*	*Spec*	*Partial*	*Strict*	*Doc*
J2SE	46	32	3	47	0	2	0	9	0	4	17	16	34
J2EE	23	8	2	32	0	2	0	14	0	2	8	6	10
JBoss	45	27	62	101	0	4	11	31	0	7	59	34	44
iText	9	11	0	4	0	4	0	0	0	0	4	2	5
Oracle	15	20	0	34	0	11	0	12	0	0	13	11	7
Total	138	383				100				283			

WebServer.java (version 2.4.11)

```
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();
}
```

WebServer.java (version 4.2.0)

```
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();
}
```

Figure 9. A confirmed bug in the JBoss application server

Table II
PRECISIONS, RECALLS, AND F-SCORES OF INFERRED SPECIFICATIONS

Library	#H	#S	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%

We manually inspected violations detected by our infrastructure. Among the violations, we identified those that we were able to determine not to be bugs. 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 bugs. Due to human factors for determining bugs, these suspected bugs may contain both false and real bugs. We further analyze the causes of these suspected bugs. Table III 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 sub-columns list the number of violations detected by the corresponding criterion. Column "*Suspected bugs*" lists the number of code snippets with suspected bugs. Similar to Column "*Violations*", sub-columns of column "*Suspected bugs*" list the number of suspected bugs 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 bugs. 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 Section VII-C for such an example). From the results in Table III, 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 Section VII-B for the discussion on the false positive rate). Second, most of the found violations are "*O/C*" and "*M/C*" violations, and we did not find any "*O/M*" violation. We suspect the reason to be that most code snippets from open source projects may have been tested by developers. As "*O/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 III, 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 Section VI for details).

To better validate the suspected bugs, we used the following procedure to determine whether they are real bugs. First, we checked the latest version of the project to determine whether a suspected bug is already fixed. If so, we deemed that we found a previously known real bug. For example, with our infrastructure, we found a suspected bug of an unclosed input stream in the JBoss application server when we use specifications inferred from J2SE's Javadoc to find

bugs. We checked JBoss's latest version and confirmed that this suspected bug is a real bug. In particular, the left code snippet of Figure 9 shows the found suspected bug, and the right code snippet of Figure 9 shows how the suspected bug is fixed. Second, if a suspected bug of a project is not fixed even in the latest version, we submitted the suspected bug as a bug report to the project's bug repository or contacted the project's developers through emails. If developers of the project confirmed that the suspected bug is a real bug, we deemed that we found a previously unknown bug. If developers of the project confirmed that a suspected bug is not a bug, we deemed it as a false bug. For those bug reports or bug-reporting emails that developers of the project have not responded yet, we deemed them as pending bugs.

Table IV shows the results. Column "*Susp.*" lists the total number of suspected bugs. Column "*Confirmed*" lists the number of confirmed bugs, and its sub-columns list the numbers of previously known and unknown bugs. Column "*False*" lists the number of suspected bugs that the developers confirmed as false bugs. Column "*Pending*" lists the number of suspected bugs whose reports developers have not responded yet. The results of Table IV show that we found 35 confirmed real bugs, including 5 previously unknown bugs. We next describe the details of some confirmed bugs and pending bugs[10].

Unclosed resources in normal code. For example, we found a previously known bug in a code snippet of the *Apache Lucene* project[11] since its developer does not close a local variable named `reader`.

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

Another previously known bug was found in the code snippet of the *Spring Framework* project[12] since its developer does not close 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);
  ...
}
```

[10]Our infrastructure found these bugs in Aug., 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.

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

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

We found a pending bug in the code snippet of the *Globus* project[13] since its developer does not close a local variable named `ix`.

```
public String ejbCreate(...) throws ...{
  try{
     Interaction ix = this.jmCon.createInteraction();
     ...
     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 this pending bug as a real bug since we found that in the same code snippet, the developer does close `ix` in another 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. For example, we found a previously unknown bug in a code snippet of the project *TopX*[14] since its developer does not close `rRset` in its *exception* block. This bug 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 pending bug in a code snippet of the *SIM-PL* project[15] 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();
   } catch (DocumentException ex) {
     throw new IOException(ex.toString());
   }
```

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

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

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

Table IV
CONFIRMATION OF SUSPECTED BUGS

Library	Susp.	Confirmed		False	Pending
		Known	*Unknown*		
J2SE	11	9	0	1	1
J2EE	16	7	0	1	8
JBoss	46	14	1	4	27
iText	4	0	0	0	4
Oracle	23	0	4	0	19
Total	100	30	5	6	59

}

In summary, using specifications inferred by Doc2Spec, we found various previously known and unknown bugs that are related to resource usages in open source projects. The results demonstrate the usefulness of our inferred specifications, because developers did produce source code that is inconsistent with the resource usages described in API documentation.

C. Threats to Validity

The threat to external validity includes the representativeness of the subjects in true practice. Although we applied our approach on Javadocs of five open source and commercial libraries, our approach is evaluated only on 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 bugs. To reduce the threat, we inspected bugs carefully and contacted developers to confirm these bugs. The threats could be further reduced by involving more experienced developers in future evaluations.

VI. BENEFITS

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.

Mining specifications from client code. Our previous work [38] 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 [38] show that in eight widely used libraries, coldspots are all more than the sums of hotspots and neutrals. Approaches that mine specifications from existing client code may have difficulties of mining specifications for those coldspots, whereas our approach does not need any client code, complementing these approaches.

Inferring specifications from comments. Tan *et al.* [37] 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 [37] to query the API documentation of J2SE 1.5, and we found only 11 exactly matched sentences. Padioleau *et al.* [31] report that iComment [37] 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.

Detecting bugs in API documentation. As shown in Table III, many false positives are caused by bugs in API documentation. For example, Figure 6a shows an inferred specification for the resource `socket@java.net.Socket`. When we used this specification to find bugs, 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 bug report to the J2SE bug database[16] (see *Bug #4118429* for details). This reported bug is confirmed as a real bug by J2SE developers. Although we count these violations as false positives in Table III, our observation suggests that our approach can also be used to find bugs in API documentation.

VII. DISCUSSION AND FUTURE WORK

In this section, we discuss various issues that are related to our approach: resource analysis, false positive rate, and extensions of our approach.

A. Resource analysis

One class/interface may have more than one resource. For each resource, our approach infers an automaton. For example, Figure 6c and Figure 6d 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[17] the descriptions of the four methods use the same common concrete noun "*JSR-77*". 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 [19]. We plan to leverage these techniques to infer better specifications in future work.

B. False positive rate

As shown in Table III, the false positive rate of our approach is 73%. It is reasonable due to four factors. First, some found bugs in documentation are interpreted as "false positive" instead of "true positive". Second, we use all inferred specifications for detecting bugs instead of using some selected specifications as some previous

[16] http://bugs.sun.com/bugdatabase/
[17] http://tinyurl.com/5zyrah

work [45] did. Third, static checkers typically produce high false positive rate (*e.g.*, 76% reported by Williams and Hollingsworth [43]). In addition, due to 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. Finally, 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.* [37]). 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 our approach takes these words into consideration, we may further reduce false positives of our approach.

C. Extensions

Bugs 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 bugs using inferred specifications, we plan to adapt our bug-detection infrastructure also for local code bases in future work. As our evaluations confirm that our approach infers various useful specifications to detect real bugs, we expect the adapted infrastructure to be useful for developers to detect bugs 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. Furthermore, for local code bases, we can use version information to improve bug detection.

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 Section III, 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.

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.

Analyzing library code. Library code analysis may help infer specifications for methods without descriptions. In particular, Fry *et al.* [15] propose 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. Library code analysis may also be useful for methods with descriptions. In particular, our previous work [47] infers specifications from library code statically. Some analysis results may be used to supplement the information from Javadocs. Library code analysis can also help find bugs either in libraries or in Javadocs as Einar and Bjarte [20] proposed.

VIII. Conclusion

Developers may still produce bugs 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 an evaluation on 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 conducted an evaluation to use inferred specifications to detect bugs. The results show that resource specifications inferred by our approach are useful to detect real bugs in practice.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. Researchers from Peking University are sponsored by the National Basic Research Program of China under Grant No. 2009CB320703, the Science Fund for Creative Research Groups of China under Grant No. 60821003, and the National Natural Science Foundation of China under Grant No.90718016. Tao Xie's work is supported in part by ARO grant W911NF-08-1-0443 and ARO grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI).

References

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proc. 32nd POPL*, pages 98–109, 2005.

[2] G. Ammons, R. Bodík, and J. Larus. Mining specifications. In *Proc. 29th POPL*, pages 4–16, 2002.

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

[4] L. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, pages 164–171, 1970.

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

[6] N. Chinchor. MUC-7 named entity task definition. In *Proc. 7th MUC*, 1997.

[7] W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: combining semi-Markov extraction processes and data integration methods. In *Proc. 10th KDD*, pages 89–98, 2004.

[8] 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.

[9] B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *Proc. 23rd OOPSLA*, pages 313–328, 2008.

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

[11] U. Dekel and J. D. Herbsleb. Reading the documentation of invoked API functions in program comprehension. In *Proc. 17th ICPC*, pages 168–177, 2009.

[12] D. Engler, D. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. 18th SOSP*, pages 57–72, 2001.

[13] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Applications of linguistic techniques for use case analysis. *Requirement Engineering*, 8(3):161–170, 2003.

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

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

[16] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *Proc. 13th ICSE*, pages 51–60, 2008.

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

[18] M. Gowri, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *Proc. 20th OOPSLA*, pages 77–96, 2005.

[19] L. Hirschman. MUC-7 coreference task definition. In *Proc. 7th MUC*, 1997.

[20] E. W. Høst and B. M. Østvold. Debugging method names. In *Proc. 23rd ECOOP*, pages 294–317, 2009.

[21] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005.

[22] L. Kof. Scenarios: Identifying missing objects and actions by means of computational linguistics. In *Proc. 15th RE*, pages 121 – 130, 2007.

[23] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proc. 7th OSDI*, pages 259–272, 2006.

[24] 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.

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

[26] D. Lo and S. Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proc. 14th FSE*, pages 265–275, 2006.

[27] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *Proc. 13th ASPLOS*, pages 329–339, 2008.

[28] A. Mikheev, M. Moens, and C. Grover. Named entity recognition without gazetteers. In *Proc. 9th EACL*, pages 1–8, 1999.

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

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

[31] Y. Padioleau, L. Tan, and Y. Zhou. Listening to programmers - Taxonomies and characteristics of comments in operating system code. In *Proc. 31st ICSE*, 2009.

[32] 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.

[33] 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.

[34] M. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proc. 29th ICSE*, pages 240–250, 2007.

[35] P. Sawyer, P. Rayson, and R. Garside. REVERE: Support for requirements synthesis from documents. *Information Systems Frontiers*, 4(3):343–353, 2002.

[36] D. Shepherd, Z. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. 6th AOSD*, pages 212–224, 2007.

[37] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or Bad Comments?*/. In *Proc. 21st SOSP*, pages 145–158, 2007.

[38] S. Thummalapenta and T. Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proc. 23rd ASE*, pages 327–336, 2008.

[39] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

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

[41] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proc. TACAS*, pages 461–476, 2005.

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

[43] C. Williams and J. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005.

[44] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *Proc. 30th ICSE*, pages 151–160, 2008.

[45] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proc. 28th ICSE*, pages 282–291, 2006.

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

[47] H. Zhong, L. Zhang, and H. Mei. Inferring specifications of object oriented APIs from API source code. In *Proc. 15th APSEC*, pages 221–228, 2008.

[48] G. Zhou and J. Su. Named entity recognition using an HMM-based chunk tagger. In *Proc. 40th ACL*, pages 473–480, 2001.