2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Automatic Model Generation from Documentation for Java API Functions

Juan Zhai[†], Jianjun Huang[‡], Shiqing Ma[‡], Xiangyu Zhang[‡], Lin Tan[§], Jianhua Zhao[†], Feng Qin[$]

[†]The State key Laboratory for Novel Software Technology at Nanjing University,

[‡]Purdue University, [§]University of Waterloo, [$]Ohio State University

zhaijuan@seg.nju.edu.cn, {huang427,ma229,xyzhang}@cs.purdue.edu, lintan@uwaterloo.ca,

zhaojh@nju.edu.cn, qin@cse.ohio-state.edu

ABSTRACT

Modern software systems are becoming increasingly complex, relying on a lot of third-party library support. Library behaviors are hence an integral part of software behaviors. Analyzing them is as important as analyzing the software itself. However, analyzing libraries is highly challenging due to the lack of source code, implementation in different languages, and complex optimizations. We observe that many Java library functions provide excellent documentation, which concisely describes the functionalities of the functions. We develop a novel technique that can construct models for Java API functions by analyzing the documentation. These models are simpler implementations in Java compared to the original ones and hence easier to analyze. More importantly, they provide the same functionalities as the original functions. Our technique successfully models 326 functions from 14 widely used Java classes. We also use these models in static taint analysis on Android apps and dynamic slicing for Java programs, demonstrating the effectiveness and efficiency of our models.

1. INTRODUCTION

Libraries are widely used in modern programming to encapsulate modular functionalities and hide platform specific details from developers. They substantially improve programmers' productivity. But on the other hand, they make software analysis very challenging. The reason is that library behavior is an integral part of software behavior such that software analysis cannot avoid analyzing library behaviors. Unfortunately, the source code of libraries may not be available. Many libraries are mixed with many languages, sometimes even in assembly code. Library implementations are usually highly optimized and full of sophisticated engineering tricks that are difficult for analysis engines to handle.

ICSE '16, May 14 - 22, 2016, Austin, TX, USA

ISBN 978-1-4503-3900-1/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884881

As an important but very challenging problem, modeling library functions has been studied by many previous works [27, 13, 21]. However, most of them require manually constructing models. This puts a lot of burden on the analysis developers. It can hardly scale to large projects that usually make use of a large number of library functions. [21] uses dynamic dependence summaries to improve analysis. However, the technique does not model the functionalities of library functions but rather their dependencies. The summary may not be accurate when conditional statements are involved and the training set is not of high quality. [23] tries to automatically generate models for library/system functions using satisfiability modulo theories (SMT) solver. The technique requires substantial manual efforts and does not scale well due to the very large search space.

According to our observation, API documentation, like Javadoc and .NET documentation, usually contains wealthy information about the library functions, such as the behaviors of a function and the exceptions the function may throw. Thus it is feasible to generate models for library functions from such API documentation. However, it is still challenging to do so since accurate linguistic analysis is needed to analyze API documentation.

This paper proposes a novel approach to generate models from Javadocs in natural languages. These models are code snippets that have the same functionalities as the original implementations but are much simpler in complexity. They can replace the original library functions during software analysis. Our contributions are highlighted as follows.

- We propose a novel and practical technique to generate substantially simplified Java code that models libraries. It allows software analysis to reason about library behaviors, without suffering from problems such as lack of source code and library implementations being too complex to analyze.
- We identify technical challenges of applying NLP techniques in modeling libraries and propose solutions to these problems.
- We implement a prototype which automatically models 326 functions in 14 commonly used Java container classes. The application of these models in Android app static taint analysis and Java dynamic slicing shows that these models precisely represent the library function behaviors and improve the efficiency and effectiveness of the analysis.

```
1 public void add(int index, E element){
2   rangeCheckForAdd(index);
3   ensureCapacityInternal(size + 1);
4   System.arrayCopy(elementData, index, elementData, index
        + 1, size - index);
5   elementData[index] = element;
6   size++;
7 }
```

Figure 1: The method *add()* in the class *ArrayList*

2. BACKGROUND AND MOTIVATION

2.1 Models in Software Analysis

Many program analyses require proper reasoning about libraries as their behaviors are an integral part of the software behavior. For example, program slicing [9], taint analysis [19] and information flow tracking [18] need to know the dependencies between input and output variables of a library function. Symbolic execution engines like [11] require precise models of libraries to construct correct symbolic constraints, and model checkers like Java PathFinder [34] need appropriate models of libraries to combat the state space explosion problem. Unfortunately, it is usually difficult and time-consuming to obtain either this kind of dependencies or the models due to the following reasons. First, the source code of library functions is often beyond reach. Even if the source code is available, it is still very challenging to analyze the code, owning to the fact that the source code tends to be prohibitively large, mix multiple programming languages, and contain substantial optimizations and engineering tricks. It is often the case that a single invocation to an API function may lead to a large number of invocations to functions internal to the library, substantially adding the complexity and cost of software analysis. Furthermore, many libraries have inherent cross-platform support, making analysis even harder.

Take the Java standard library as an example. The Java Development Kit (JDK) contains the Java standard library source code (version 8.0), the size of which is 81.7MB. There are more than 10000 classes and 80000 methods. To achieve the binary level compatibility of native library methods across all Java virtual machine (JVM) implementations on a given platform, JDK libraries invoke native methods through the Java Native Interface (JNI) framework. These native methods are implemented in other languages such as C++ and assembly, and their source code is unavailable in general. Besides, there are multiple implementations for such a method on different architectures and JVMs. Fig. 1 shows a simple library function *add (int index, E element)* in the *ArrayList* class. There are three function calls, including *System.arrayCopy()*, in five effective lines of code (eLOC). The implementation of the function *System.arrayCopy()* is unavailable in the source code folder. It is implemented in native code using JNI. From this, we can see that analyzing JDK functions is challenging.

Due to these reasons, library function models are usually constructed and provided to replace the original implementation during analysis. These models are code snippets that have the same functionality as the original library functions but are much simpler. They can be used in place of the original functions during software analysis. Note that even though they are not as efficient or sophisticated as the original functions, they are much easier to analyze. Currently, the majority of models used are manually constructed [27, 11], which represents a substantial burden for the analysis developers due to the large number of library functions and their rapid evolution. Furthermore, library functions are often optimized to achieve high performance. The details of these optimizations are usually not the essential part of the functional models. For example, as shown in Fig 2, the model of the *add()* function can be represented by simple array operations, guarded by a range check predicate. The low level details of *System.arrayCopy()* and *rangeCheckForAdd()* are not needed in the model. All these factors motivate us to develop an approach to automatically generate models for library functions.

2.2 Approach Overview

It is a common practice for library developers to provide behavior description of library functions in natural language in the Application Programming Interface (API) documents. J2SE's Javadoc [3] is a typical example of such API documentation, which offers wealthy information such as class/interface hierarchies and method description. Our idea is hence to construct models from API documentation.

In recent years, natural language processing (NLP) techniques have made tremendous progress and have been shown to be fairly accurate in identifying the grammatical structure of a natural language sentence [16, 17, 8, 35]. This enables us to leverage NLP techniques including *Word Tagging/Part-Of-Speech (POS) Tagging* [16], *Phrase and Clause Parsing (Chunking)* [16] and *Syntactic Parsing* [15] to acquire semantics of sentences in API documentation.

Basic Idea: Given the documentation of a Java API function, we leverage an NLP tool to generate the parse trees of the individual sentences. For each sentence, we match its parse tree with the tree patterns of a set of predefined primitives, which are small code snippets implementing very basic functionalities. In particular, we try to identify a subset of primitives whose tree patterns can cover the entire parse tree when they are put together. The parameters in the primitives are also instantiated by the corresponding nodes in the parse tree. The same procedure is repeated until the parse trees of all sentences are completely covered. The composition of the code snippets corresponding to the identified primitives produces a model candidate. Since there are multiple possible parse trees for a sentence and many ways to cover a tree, multiple candidates are generated. The invalid ones are filtered out by testing if a candidate behaves differently from the original library function.

Example. Fig. 2(a) gives part of the documentation of the library function *add (int index, E element)* in Fig. 1. The text can be divided into four parts: (1) the declaration of the function, including explanations of the parameters (shown as box ④ in the figure); (2) the functionality of the method (boxes ② and ③); (3) the exception handling logic (box ⑤). We take box ② as an example to demonstrate our idea. The parse tree generated by an off-the-shelf NLP tool is shown in Fig. 2 (b). We perform further processing on the tree, including transforming the tree structure to accommodate ambiguity and recognizing parameters, to produce a tree-like *intermediate representation* (IR) as shown in Fig. 2 (c). We then try to use the primitive tree patterns to create a tiling of the IR. In this case, the tree template of the `insert` primitive (as shown in Fig. 2 (d)) alone can cover the whole IR tree. As such, the model code for the sentence is gener-

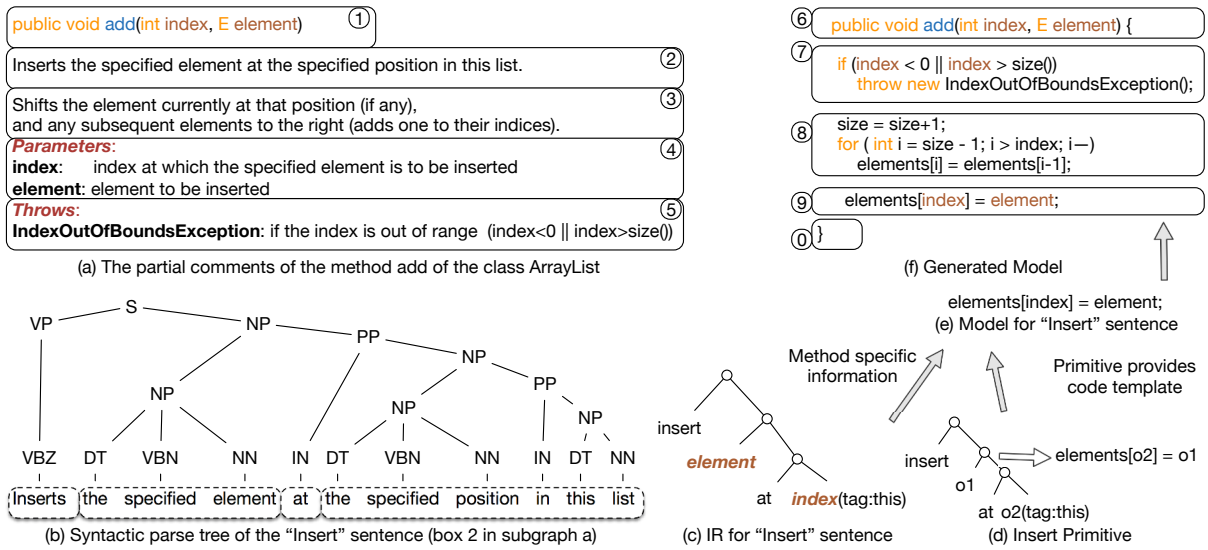


Figure 2: Motivation example.

ated as in ⑨. Note that the parse tree nodes that denote the variable names (i.e., "`element`" and "`index`") allow us to instantiate the parameters o_1 and o_2 in the primitive. The code for the other sentence and the exception handling description is similarly generated. Note that boxes ⑧ and ⑨ have a different order than that in the text. Our technique generates multiple candidates including those with different orders and use testing to prune out the invalid ones.

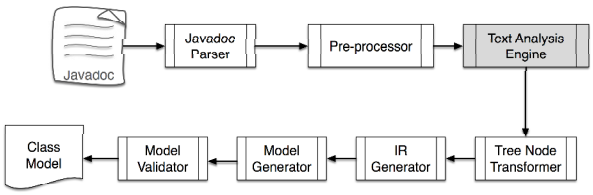


Figure 3: Overview of our solution

3. DESIGN

Fig. 3 gives the overarching design of our approach. The whole system takes a Javadoc as input, and uses the Javadoc parser, pre-processor, text analysis engine, tree transformer, intermediate representation (IR) generator, model generator and model validator to generate models. In particular, the Javadoc parser takes the Javadoc in a structured HTML format as input and extracts contents from both class and method description. The pre-processor performs some synonym analysis and enhances the extracted sentences. Then a tree structure is generated for each sentence by the text analysis engine which leverages Stanford Parser [16, 24] and domain specific tags to perform the natural language processing (NLP). After that, the tree transformer automatically generates variants for some of those tree structures. These variants represent the different interpretations of the sentence in the context of Java programming. The variants as well as the original tree structure are processed by the IR generator, which identifies and marks function parameters in each tree structure. The model generator searches for tilings of the IRs using the tree patterns of the pre-defined primitives, and eventually generates multiple model candidates for each function. These candidates are passed to the model validator to filter out candidates that behave differently from the original function. The model validator makes use of Randoop [20] to automatically generate test cases. The first candidate that passes all the test cases is the resulting class model.

3.1 Pre-processor

The pre-processor accepts the descriptions extracted by the Javadoc parser and performs three kinds of analysis.

Table 1: The documentation of the method *indexOf* of class *ArrayList*

`public int` *indexOf* `(Object o)`
Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. More formally, returns the lowest index `i` such that `(o==null ? get(i)==null : o.equals(get(i)))`, or -1 if there is no such index.
Specified by:
indexOf in interface `List<E>`
Overrides:
indexOf in class `AbstractList<E>`
Parameters:
`o` - element to search for
Returns:
the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

Equivalence Analysis. Our pre-processor classifies words into pre-defined semantic classes based on domain dictionaries. For example, the words "`adds`" and "`inserts`" are semantically equivalent in method description. Classifying them into one category can relieve the effort of identifying mappings for each word in the generated IRs.

Redundant Information Elimination. We attempt to remove sentences that are used to elaborate other sentences. Just to name a few, sentences starting with "`in other words`", "`namely`", "`More formally`", and "`That is to say`" are explanations of other sentences, and they will be eliminated in our system to prevent redundancy in analysis.

Sentence Augmentation. The sentences in return descriptions and exception descriptions in Javadocs tend to be incomplete, which makes it difficult for the Stanford parser to analyze. Our sentence augmentation component aims to enhance those sentences for easy parsing.

For return descriptions, the verb "`return`" is often omit-

ted. For example, "`true if this list contained the specified element`" is the return description for the *remove (Object o)* method in the *ArrayList* class. In this case, our approach checks whether the sentence is under the tag "`return`" in the corresponding Javadoc. If so, the verb "`Returns`" is added to the sentence.

For exception descriptions, the throw behavior and the exception thrown are typically omitted. For example, "`if this vector is empty`" is the exception description for the *lastElement ()* method in the *Vector* class. The verb `throw` and the exception `NoSuchElementException` are missing. In this case, our approach checks whether the sentence is under the tag "`throw`" in the corresponding Javadoc. If so, the phrase "`Throws NoSuchElementException`" is added to the sentence to augment the original sentence. Note that the exception name can be easily extracted and used in the augmentation (e.g. in Fig. 2 (a)).

3.2 Text Analysis Engine

The text analysis engine generates a grammatical tree structure for each pre-processed sentence through natural language processing. To do this, we leverage the state-of-the-art Stanford Parser with domain specific tags.

The Stanford parser parses a sentence and determines POS tags associated with different words and phrases. These tags are essential to the generation of the syntactic tree structure for the sentence. There are some words that represent nouns in programming while representing adjectives in general linguistics. For instance, in the sentence "`Returns true if this list contains the specified element.`", "`true`" should be regarded as a noun in programming rather than an adjective. But the Stanford parser would identify it as an adjective. In addition, the Stanford parser may incorrectly identify some words as nouns while in fact they should be marked as verbs. For example, "`Returns`" in the mentioned description is a verb, but it is incorrectly identified as a noun by the Stanford parser. Therefore, a *POS restricting* component is added to the Stanford parser to force it to use our pre-defined tags for some programming-specific words. Some pre-defined tags are as follows:

- noun: true/false/null
- verb: returns/sets/maps/copies
- adjective: reverse/next/more/empty

3.3 Tree Transformer

The tree transformer transforms the original tree structure generated by the Stanford parser to produce variants. Each variant corresponds to a different interpretation of the sentence. We need to generate multiple interpretations of a sentence due to ambiguities in natural languages, and consequently we will generate multiple model candidates which are passed to the model validator discussed in Section 3.6 to filter out incorrect models. Text analysis engines like the Stanford parser are able to generate multiple trees with different semantics. Unfortunately these parsers cannot understand the real semantic of a sentence, especially when domain knowledge is involved. The Stanford parser can generate k parse trees for a sentence with k given by the user. However, there is no way for the parser to guarantee that the tree with the expected meaning is generated for a sentence even with a large k.

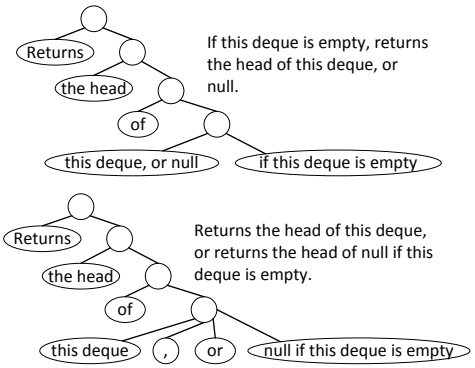


Figure 4: Syntactic trees with unexpected meanings

Take the method description "`Returns the head of this deque, or null if this deque is empty`" as an example and we set the value of k as 20. None of the 20 parse trees generated by the Stanford parser gives the exact meaning conveyed by this sentence in the context of programming, which should be "`Returns the head of this deque, or returns null if this deque is empty`". Two of the trees with unexpected meanings are shown in Fig. 4. Some subtrees are summarized to be one tree node to save space. For example, the left tree expresses "`If this deque is empty, returns the head of this deque, or null`" while the right tree conveys "`Returns the head of this deque, or returns the head of null if this deque is empty`".

If the value of k is large, a lot of computation will be wasted in generating and analyzing trees with incorrect meanings. Through an analysis of the generated trees, we found that in most cases, the Stanford parser is good at recognizing individual phrases of a sentence in the context of programming, and the places where ambiguity arises are those phrases starting with "`, or`" and "`, and`". We also discovered that by lifting up or pushing down the node "`, or`" or "`, and`" and its right siblings for only a few number of times in the tree produced by the Stanford parser[1], we can get the tree which conveys the expected meaning of the sentences.

Thus we propose Algorithm 1 to transform the parse tree from the Stanford parser to produce a set of variants by repositioning only the conjunctive nodes. This algorithm takes the root of the tree generated by the Stanford parser as an input and produces a set of tree variants, represented by *variantSet*.

First, *variantSet* is initialized to contain only the original tree *root* (lines 1-2). Then, the main loop is executed to generate variants (lines 3-11). In each iteration, lines 5-7 make a transformation of each tree in *variantSet* to get variants and add them back to *variantSet*. The transformation process is the function *transform()* shown in Fig. 5. When the set does not change any more (line 8), a fix-point is reached, meaning all possible variants have been identified, and the process terminates.

As can be seen in Fig. 5, for each tree node n, if it is a node representing "`,`" and its right sibling node represents "`or`" or "`and`", the function transforms the tree by lifting up and pushing down the node n and all its right siblings. The

[1]The parser by default returns a single tree that it considers having the highest probability of denoting the real semantics of the sentence.

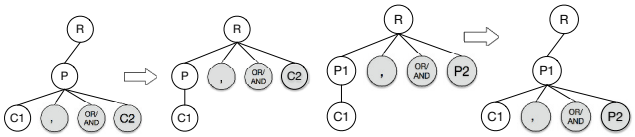
$$transform(r,n) = \bigcup_{c}^{c \in Children(n)} transform(r,c) \cup \begin{cases} liftUp(r,n) \cup pushDown(r,n) & : \text{a ``,'' node followed by a ``or'' or ``and'' node} \\ \emptyset & : \text{otherwise} \end{cases}$$

Figure 5: Function transformation

Algorithm 1 Transforming one Tree Node: function *transformTree*

Input: *root* - root of the original tree node
Output: *variantSet* - a set of variants of the original tree node

```
1: variantSet ← ∅
2: variantSet ← variantSet ∪ root
3: while true do
4:    oldSet ← variantSet
5:    for all tree ∈ variantSet do
6:       variantSet ← variantSet ∪ transform(tree, tree)
7:    end for
8:    if variantSet == oldSet then
9:       break
10:   end if
11: end while
12: return variantSet;
```



(a) Lifting up. (b) Pushing down.

Figure 6: Lifting up and pushing down nodes. The shaded nodes are repositioned.

lifting up operation is shown in Fig. 6(a) while the pushing down operation is shown in Fig. 6(b). In other cases, the tree keeps unchanged.

Consider the documentation in Table 1. The left tree in Fig. 7 is generated by the Stanford parser for the sentence "`Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element`". From this tree, we get the semantics "`Returns the index of the first occurrence of the specified element in this list, or returns the index of the first occurrence of -1 if this list does not contain the element`". which is not the expected meaning of this comment sentence. By lifting up the nodes "`,`", "`or`", and "`-1 if this list does not contain the element`" on the seventh layer of the left tree five times, we get the tree on the right, which conveys the exact meaning of this comment. Note that since our tool does not know which variant represents the intended meaning, it generates all of them and then selects the right one through testing. From Fig. 7, we can see that the number of variants generated by lifting up the left tree is five which is acceptable.

3.4 Intermediate Representation Generator

Our intermediate representation generator manipulates trees generated by the tree transformer to constructs IRs by substituting subtrees, identifying parameters, and adding labels based on programming domain knowledge. We cannot directly translate a generated tree to code unless we associate tree nodes with code artifacts. To achieve this, our IR generator performs two major tasks: (1) Parameter recognition, which identifies parameters; (2) Loop and conditional structure recognition.

Parameter Recognition: This component recognizes parameters in method descriptions. Javadocs refer to parameters using several different descriptions which are summarized as patterns shown in column "Pattern" in Table 2. However, due to the complexity and ambiguity of natural languages, these patterns may not always imply parameters. In some cases, even the occurrences of the same word as the parameter name do not indicate the corresponding parameter. For example, the description of the method *set(int index, E element)* in the class *ArrayList* is "`replaces the element at the specified position in this list with the specified element`". The phase '`the element`' does not refer to the parameter *element*, but '`the specified element`' does. Our solution is again to generate all possibilities and let the model validator to determine the right mappings (of parse tree nodes to parameters).

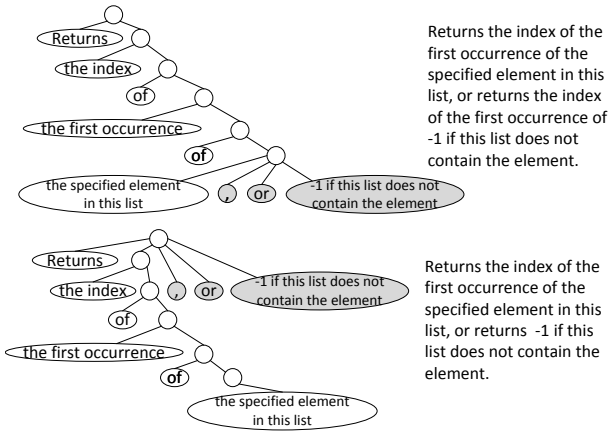


Figure 7: Lifting up nodes example.

Algorithm 2 describes the process of recognizing placeholders of parameters in method description. It takes three arguments, i.e., all the parameters of the method being modeled *params*, the root of the parse tree *root*, and a predefined list of synonyms *synonyms*. The output of this algorithm is represented as *irSet*, which is a set of IRs generated by recognizing parameters of the tree. The algorithm works as follows. The first step is to initialize *irSet* to contain the original tree *root* (lines 1-2). Next, the main loop is executed to recognize parameters in *root* (lines 3-9). The algorithm recognizes the parameters one by one with each iteration responsible for recognizing one parameter on the tree(s). Since there are multiple possible places that indicate a parameter, multiple trees may be generated by associating the parameter with different tree nodes. Each iteration in the loop 5-6 handles one tree *ir* from the previous round. The resulting trees are stored in *newSet*. Recognizing a variable on a tree is done through a recursive function *traverse()* shown in Fig. 8, which tries to match the parameter with each tree node. Since the parameter can match multiple nodes (corresponding to that there are multiple words that seem to mean the parameter), the function may produce multiple trees, each representing one possible match. At the end, each tree in *irSet* is a tree with recognized parameters and each tree node represents at most one parameter.

Algorithm 2 Identifying Parameters in one Tree: function *identifyParams*

Input: *params* - all the parameters of the method being modeled
root - root of the original tree node
synonyms - the predefined list of synonyms
Output: *irSet* - a set of IRs after recognizing parameters

```
1: irSet ← ∅
2: irSet ← irSet ∪ root
3: for all param ∈ params do
4:    newSet ← ∅
5:    for all ir ∈ irSet do
6:       newSet ← newSet ∪ traverse(ir, param)
7:    end for
8:    irSet ← newSet
9: end for
10: return irSet;
```

$$traverse(t,p) = \bigcup_{c}^{c \in Children(t)} traverse(c,p) \cup recParam(t,p)$$

Figure 8: The function traverse

Function *recParam()* in Fig. 8 is to recognize a parameter by pattern matching. The patterns are described in Table 2. Particularly, if a subtree rooted at t matches a tree pattern in the first column and satisfies the condition in the second column, t is replaced with a node representing the parameter. The third column shows some examples. For example, the rule in the first row means that, if a subtree denoting a phase "`the` w_1 w_2" is observed and the concatenation of w_1 and w_2 is a synonym of the parameter name. The subtree is replaced with a node representing the parameter. Other rules are similarly defined.

It is worthy mentioning that we take special care of the word "`this`" as it often has special meanings in our context. In particular, for every occurrence of phrase "`this WORD`" with WORD being the class name or its abbreviation, we label the corresponding tree node with a special tag *this*, indicating the code (to be generated) operates on the receiver object. For example, "`this list`" is used to represent the receiver *ArrayList* object in the documentation of *ArrayList*.

Take the right tree in Fig. 7 as an example. According to the second rule, our approach substitutes the subtree representing "`the specified element`" with a node representing variable *o*, which is further tagged with "`this`" due to the phrase "`in this list`".

Structure Recognition: This component recognizes programming structures indicated in method descriptions. Documentation descriptions use some special words to specify how the behavior is carried out, such as the condition under which the behavior will execute and how many times the behavior should execute. These restrictions are projected to programming structures, like loops and conditionals, which are vital for generating models. Our approach recognizes these structures by recognizing subtrees containing the special words, and substituting subtrees as well as adding tags.

Loop structure. Plurals and singular nouns modified by "`each`" tend to imply that the behavior should be executed for multiple times, which indicates a loop structure. For example, the phrase "`all of the elements`" in the sentence "`Inserts all of the elements in the specified collection into this list, starting at the specified position`" of the method *addAll(int index, Collection<? extends E> c)* in *ArrayList*, indicates that the insert operation should execute multiple times. Thus the model for this behavior must have a loop structure. In this case, our approach adds a *loop* tag to the IR and substitutes the subtree representing "`all of the elements`" with a node representing "`elements`". Some phrases in natural language do not explicitly indicate a loop structure. Instead, they can imply the iteration order. For example, the subtree representing "`the first occurrence of`" in Fig. 9(a) implies that if there is a loop structure for this behavior, it should iterate from left to right. In this case, our approach adds a *ltr* tag to the IR and trims the subtree representing "`the first occurrence of`".

Conditional structure. Words like "`if`" and "`when`" in natural languages indicate a conditional statement in programming while words like "`otherwise`" indicate an else branch. Our approach adds tags *if* and *else* to the subtree that is modified by these words. For example in Fig. 9(a), an *if* tag is added to the subtree representing "`-1`". In addition, our technique recognizes whether the description is affirmative or negative to determine the condition of the *if* statement. For example, the behavior "`contain`" in Fig. 9(a) is modified by "`does not`". It means that the condition of the *if* statement should be the negation of the result of the contains behavior. In such cases, our approach trims the subtree representing "`does not`" and adds a flag "-" to the node representing "`contain`". For instance, we get the IR in Fig. 9(b) by recognizing structures of the IR in Fig. 9(a).

3.5 Model Generator

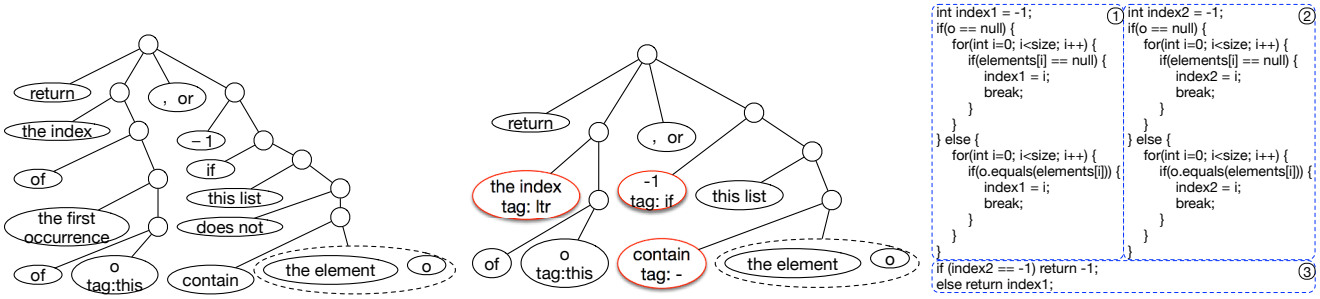
In this section, we introduce our method of generating models based on IRs. For each IR, a tiling by the tree patterns of primitives is identified. The corresponding code snippets of the primitives are assembled to constitute the model of the IR. The code snippets for all the IRs in a method description are further integrated to gain the method model. The class model is eventually generated based on the individual method models and the class information collected earlier. Since one sentence can have multiple IRs, multiple method/class model candidates are generated.

For all the Java container classes (e.g. lists, queues, and stacks), we observe that it is sufficient to use a one-dimensional array to model them. As such, many of our primitives are essentially operations manipulating the underlying one-dimensional array. Using primitives avoids generating models from simple and low-level expressions and statements, which requires exploring a very large search space for the proper combination. Table 3 presents part of the pre-defined primitives. Each primitive has a *tree pattern* and a piece of code template (on the underlying one-dimensional array). The tree pattern describes the syntactic structure of the description of the primitive. It is used to cover part of an IR tree to create a tiling.

Each cell in the table includes the name of the primitive, the description, and the tree pattern. The corresponding code templates are elided. Each primitive has a set of parameters, which are also denoted in the tree pattern. The primitive *insert*(o_1, o_2) represents the functionality of inserting an object o_1 to o_2. The corresponding code template implements this functionality on the one-dimensional array. Its tree pattern essentially describes how such functionality is expressed in a natural language and hence can be used for IR tree tiling. We also have other primitives such as *copy* and *apply*. We have a total of 12 primitives.

Table 2: Patterns for Labeling a Parameter n. Symbols w, w_1 and w_2 denote words.

Pattern	Condition	Example
the w_1 w_2	w_1, w_2 and n are synonymous	*setSize(int **newSize**)* in *StringBuffer*: "*Throws ArrayIndexOutOfBoundsException if the **new size** is negative*".
the *specified* w	w and n are synonymous	*get(int **index**)* in *ArrayList*: "*Returns the element at **the specified position** in this list*".
the w *specified*	w and n are synonymous	*add(int **index**, Attribute object)* in *AttributeList*: "*Inserts the attribute specified as an element at **the position specified***".
the *specified* w	w == the type of n	*add(**Attribute** object)* in *AttributeList*: "*Adds **the Attribute specified** as the last element of the list*".
the w *specified*	w == the type of n	*append(**StringBuffer** sb)* in *StringBuffer*: "*Appends **the specified StringBuffer** to this sequence*".
the w *argument*	w == the type of n	*append(**char** c)* in *StringBuffer*: "*Appends the string representation of **the char argument** to this sequence*".
the *argument*		*removeElement(Object obj)* in *Vector*: "*Removes the first (lowest-indexed) occurrence of **the argument** from this vector*".
the w	w and n are synonymous	*set(int index, E **element**)* in *ArrayList*: "*Replaces **the element** at the specified position in this list with the specified element*".



(a) IRs after parameter recognition (b) IRs after structure recognition (c) Model

Figure 9: IRs and Model.

The tiling algorithm is very similar to that used in compiler code generation [10]. It is a greedy algorithm that tries to cover an IR tree from the root to the leaves. Particularly, it first tries to cover a subtree starting from the root (of the original tree). It then tries to cover the remaining parts of the tree, until a tiling is found.

Example. Consider the IR tree in Fig. 9(b). The algorithm first covers the top part of the tree with the pattern of the primitive *return*, which returns different values depending on a condition. It then covers the left sub-tree with the *indexOf* primitive and the right sub-tree with the *contain* primitive. The tiling is shown in Fig. 10. The resulting code is shown in Fig. 9 with the order of ①, ②, and then ③. Observe that the code generation is bottom up: boxes ① and ② for the *indexOf* and *contain* primitives, respectively, and then box ③ is for *return*. Also observe that although the generated model functions correctly, it is a bit redundant as the code templates for the first two primitives are essentially the same. This redundancy is easily precluded through a post-processing step.

We will assemble code snippets of each IR to form the method model, during which procedure, our technique also explores the different orders of the code snippets.

3.6 Model Validator

As pointed out before, our approach can generate multiple model candidates due to the ambiguity of natural languages and the limitation of current NLP tools. But most of them have inconsistent behaviors with the original library, which

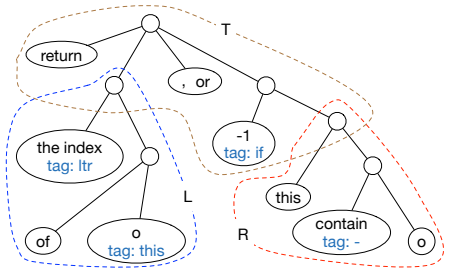


Figure 10: Tiling an intermediate representation.

makes it necessary to filter out the inconsistent model candidates. Our model validator accepts multiple model candidates and excludes the candidates that behave differently from the original method. We leverage the existing work, Randoop [20], which can automatically generate test suites for Java classes to help us check whether the generated models preserve the behavior of the original method(s). For each class, we generate 720 test cases on average.

4. EVALUATION

We have implemented a prototype and conducted a set of experiments to evaluate the prototype. In our evaluation, we focus on two aspects:

1. The effectiveness of the model generation technique.
2. Using the generated models in other analyses.

Table 3: Primitives

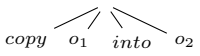
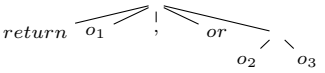
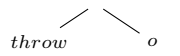
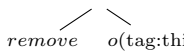
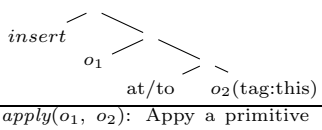
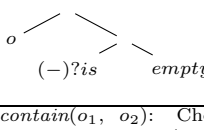
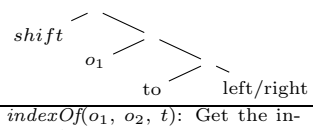
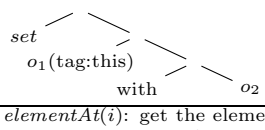
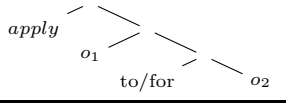
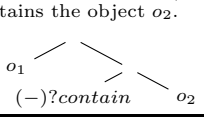
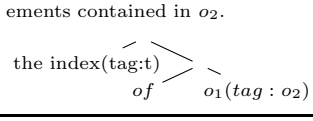
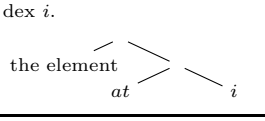
copy(o_1, o_2): Copy the content specified by o_1 into the destination o_2. 	*return*(o_1, o_2, o_3): Return o_2 if condition o_3 holds, o_1 otherwise. 	*throw*(o): Throw an exception specified by o. 	*remove*(o): Remove object o from the container. 
insert(o_1, o_2): Insert object o_1 at a location specified by o_2. 	*isEmpty*(o): Check whether o is (not) an empty container. 	*shift*(o): Left/right shift an object o. 	*set*(o_1, o_2): Set a field o_1 of the receiver object to o_2. 
apply(o_1, o_2): Appy a primitive operation o_1 to an object o_2. 	*contain*(o_1, o_2): Check whether the container o_1 (does not) contains the object o_2. 	*indexOf*(o_1, o_2, t): Get the index of o_1 in o_2 by searching elements contained in o_2. 	*elementAt*(i): get the element at the position specified by index i. 

Table 4: Overall Result

Class	#T	#M	%	#C	GT	VT	#CN
ArrayList	34	29	85.29%	128	5.85	13876	490
Vector	54	46	85.19%	512	8.22	52833	500
Stack	6	5	83.33%	1	2.52	102	1
ArrayDeque	36	35	97.22%	64	7.26	6922	756
LinkedList	42	41	97.62%	8192	8.12	877281	545
HashMap	28	23	82.14%	128	5.85	14447	1337
LinkedHashMap	15	14	93.33%	1	4.35	108	667
HashSet	13	12	92.31%	1	3.06	107	726
LinkedHashSet	5	4	80.00%	1	2.50	107	641
AttributeList	15	11	73.33%	2	2.57	218	638
RoleList	14	9	64.29%	2	2.09	109	836
RoleUnresolvedList	14	9	64.29%	1	1.81	109	886
StringBuffer	54	40	74.07%	1	14.81	107	950
StringBuilder	54	40	74.07%	1	13.51	109	1098
Summary	397	326	82.12%				

If not specified in the following sections, the evaluation was conducted on a machine with Intel(R) Core^{TM}i7-3770 CPU (3.4 GHz) and 8GB main memory. The operating system is Ubuntu 12.04, and the JDK version is 7.

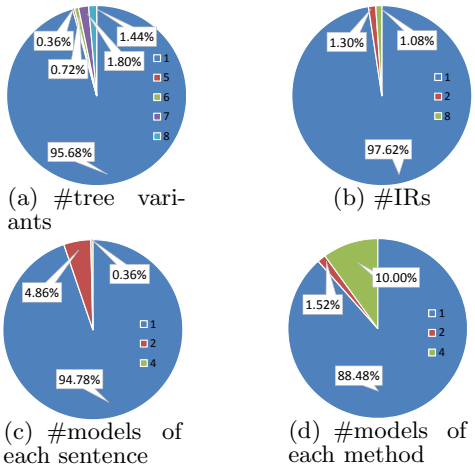
4.1 Effectiveness in Model Generation

To evaluate the effectiveness of our model generation, we applied it to 14 Java container-like classes.

Table 4 shows the results. Column "Class" lists the names of the modeled classes, which are grouped by different packages. The packages from top to bottom are respectively *java.util*, *javax.management*, *javax.management.relation*, and *java.lang*. For each class, column "#T" lists the total number of methods of the original JDK class. Column "#M" lists the number of methods that our tool can successfully model. Here the models are functionally equivalent to the original methods. Column "%" lists the ratio of modeled methods to total methods. Column "#C" lists the number of model candidates for each class. Column "GT" lists the time used to generate class model candidates in seconds. Column "VT" lists the time used to validate class models using Randoop in seconds. Column "#CN" lists the average number of test cases generated by Randoop. Row "Summary" lists the *total* numbers for columns "#T" and "#M", and gives the *average* of column "%".

From the results in Table 4, we have the following observations. First, we can generate models for most methods in these classes, which indicates the effectiveness of our approach. For the last five classes, the percentage is relatively low. The low ratios of the classes *AttributeList*, *RoleList* and *RoleUnresolvedList* result from the incompleteness of their documentations. For example, the method *add(int index, Object element)* in *AttributeList* should throw an exception (*java.lang.IndexOutOfBoundsException*) when *index* is less than 0 or greater than the length of this list. But no sentences describe this behavior which makes it impossible for our tool to generate the corresponding code. Our approach handles each sentence separately while descriptions of some methods use several sentences to depict one primitive behavior. This leads to the low ratios of the classes *StringBuffer*, and *StringBuilder*. Take the method *insert(int index, char[] str, int offset, int len)* of *StringBuffer* as an example. The insertion operation is described using the following three sentences: "`Inserts the string representation of a subarray of the str array argument into this sequence. The subarray begins at the specified offset and extends len chars. The characters of the subarray are inserted into this sequence at the position indicated by index`". Currently, our approach cannot handle such cases. We plan to correlate multiple sentences in the future. Second, both the time used to generate models and the time used to verify models are acceptable. The time used to generate models for *StringBuffer* and *StringBuilder* are much longer than that of the other classes, which results from the fact that there are much more sentences to be analyzed in these two classes. Much of the time used to verify models is spent in generating test cases by Randoop. The average time used to generate a series of test cases for one class model is 108 seconds. Third, the validation time and the number of the candidates has linear relationship.

The pie charts in Fig 11 show more statistics. Fig. 11(a) gives the distribution of the tree variants derived from a tree due to the lift-up and push-down transformations. Each tree corresponds to one sentence. Fig. 11(b) presents the distribution of the number of IRs derived from a tree after parameter recognition. Fig. 11(c) shows the distribution of the numbers of models generated for a sentence. From these three pie charts, we can see that the number of generated models of each sentence is not simply the product of the number of tree variants and the number of IRs of each tree variant. The reason is that we cannot generate models for some IRs which cannot be tiled with our proposed primitives. Fig. 11(d) shows the distribution of the number of model candidates generated for each method. Only one model is generated for a method in the majority



(a) #tree variants (b) #IRs

(c) #models of each sentence (d) #models of each method

Figure 11: Distributions

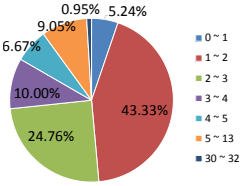


Figure 12: Line of code comparison with JDK

cases(88.48%), and the maximum number of models for a method is 4 with a percentage of 1.52%, which is acceptable for the validation step.

Fig. 12 shows the comparison of the line of code between our models and the original JDK. We counted the line of code for all functions including its dependencies, and divided the line of code of the original JDK by ours. From the chart we can see that for normal cases, the original JDK is 1 to 13 times larger than our models. There are some cases that our models are larger because some JDK methods calls JNI functions like *System.arrayCopy()*, while our models are completely implemented in Java. And there are some extremal cases that the code size is large, such as the model for *HashMap.clone()*, which needs to check boundaries and clone all objects inside it.

Fig. 13 shows the distribution of the appearances of the primitives in documentation. We observe that *add*, *remove* and *shift* take a large portion because our models focus on container classes. The primitive *throw* takes a large portion as well because Java API document clearly defines this behavior for many functions.

Fig. 14 shows the number of primitives used to model each method. Most methods can be constructed by only a few primitives.

4.2 Our Models in Static Taint Analysis

To evaluate the effectiveness and efficiency of our models in static analysis, we conduct a taint analysis on Android that detects the undesirable disclosures of user inputs to

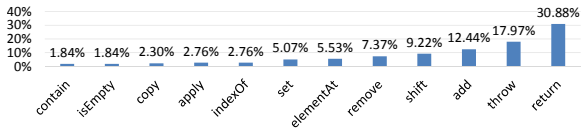


Figure 13: Distribution of primitives in documents

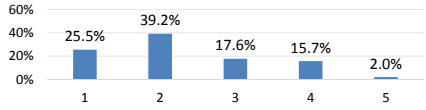


Figure 14: # of primitives per method model

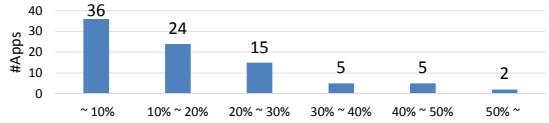


Figure 15: Efficiency improvement distribution

public channels on 96 apps that were previously known to have user input disclosures. We limit the sinks to Internet access and Log writing, and set a 30 minutes timeout for each app in our experiment. We first run the analysis with the original JDK source code and then replace part of the JDK with our models and then compare the number of information leak warnings reported and the performance.

Among the 96 apps, 1 app does not have any use of the modeled methods and is hence excluded; 8 other apps are also eliminated because they time out or run out of memory using both our models and the JDK implementation. The results of the remaining 87 apps are summarized as follows.

Effectiveness. The analysis reports the same set of information leak warnings for both versions for almost all apps, except app *com.yes123.mobile* which is reported to have 16 paths using our models and 14 paths using JDK. We manually inspect the differences and find that the two paths contain invocations of *ArrayList.toArray(Object[])*. The JDK implementation calls a JNI method *System.arrayCopy()* to copy data from the 1st argument to the 3rd argument. Our model provides an implementation of the functionality such that static analysis is not blocked by the absence of Java code. We have also found 51 of the 87 apps have similar JNI calls such that the static analysis cannot handle those calls properly. However, since those calls are not on any leak path from the source to the sink, they do not induce differences in the bug report.

Efficiency. Fig. 15 shows the performance comparison by presenting the distribution of the improvement. The performance improves more than 10% for 51 apps by using our model. The maximum improvement is 50% and the average is 16%, even with a small portion of the JDK library replaced by our models.

4.3 Our Models in Dynamic Slicing

In addition to the static taint analysis on Android applications discussed in Section 4.2, we also evaluate our models in dynamic program analysis techniques, namely, Java dynamic slicing. We utilize JavaSlicer [4] and 5 benchmark programs including SPECJBB [7], FunkyJFilter [2], ListAppend [6], batik in DaCapo [1], and some unit tests with JTreg [5]. JavaSlicer is a dynamic slicing tool. We first run the program with JavaSlicer and inputs. During runtime, JavaSlicer collects traces of the execution of this program. Then for each benchmark program, we choose one variable as the input, and JavaSlicer reports the number of unique Java bytecode instructions and all the detailed instructions in the dynamic slice for this given variable. Note that an instruction that gets executed multiple times is reported only once. For all the 5 benchmark programs, we run them with parameters given as examples in their manuals, and choose the generated result value as the starting point of the slicing.

Table 5: Dynamic slicing results

	Naive approach		Our model	
	slice size	time	slice size	time
SPECJBB	564	1.76	393	0.73
FunkyJFilter	629	2.18	5	2.16
ListAppend	7,050	12.79	504	5.9
Batik	35,721	1,973.65	5,516	295.77
Unit Tests	32	0.39	3	0.36

All experiments are conducted successfully.

To demonstrate the results, we compare our models with a naive approach that considers any output of a method depends on all the inputs. The results are shown in Table 5. The first column lists the names of the benchmark programs. For each program, we show the size of the slice and the running time of using the naive approach (column 2 and 3, respectively), and using our models (column 4 and 5, respectively). As shown in the table, for all the programs, JavaSlicer produces slices of smaller size with our models than the naive approach. Due to the characters of these different benchmarks, we get different results. On average, the slice size is 32 times smaller due to the higher precision of our models. For the same reason, it takes less time to produce slices using our models than using the naive approach.

5. LIMITATIONS

Our technique heavily depends on the quality of documentation. If the documentation is incomplete or uses strange syntax or wordings, our tool may not generate the correct models. This is a general limitation for many NLP based techniques [37, 30]. Our technique relies on a set of syntactic patterns to recognize parameters and primitive operations. These patterns are mainly derived from Javedocs. They may not be generally applicable. However, we argue that our technique is general in principle and it is a valuable step towards automated model generation, which is a hard problem in general. We envision the writing style of documentation may not change as frequently as the library implementation. As a result, our technique can serve as an automated approach to quickly generate models for a large number of library functions, as demonstrated by our results. In the future, it may be a beneficial practice to enforce a fixed documentation style.

Furthermore, our current study largely focuses on container-like libraries as they are the most widely used category. It is very difficult to model the precise functionalities of some special-purpose libraries such as math libraries. However, we also observe that having the precise models for those libraries are unnecessary in many applications. For example, the input-output dependencies of most math library functions are very simple, despite their complex computation.

It is also possible that the test cases used in model validation are not sufficient so that we admit some incorrect models. Even though we have not encountered such problems in our experience, it would be interesting to use more rigorous validation techniques such as equivalence checking.

6. RELATED WORK

Our approach is related to previous works closely on two areas: documentation analysis and environment modeling.

Documentation analysis. Previous researchers analyze natural language documents for many purposes. [22] proposes methods to infer method specifications from API descriptions. [37, 14] try to detect code-document inconsistencies by leveraging NLP and program analysis. [36] gives programmers suggestions for usage of APIs based on mining the usage patterns. [28, 25] analyze bug reports to remove duplicates. [26] tries to help generate use cases in real world development by inspecting language documents. [30, 29] use comments with other techniques to detect inconsistencies between comments and code. In contrast to these approaches, our approach aims to help program analysis by constructing models for libraries. We only use Javadocs, but the approach can be easily expanded to other types of natural language documents, e.g. comments, bug reports.

Environment modeling. To reduce manual efforts for environment models, various approaches [11, 27] have been proposed. [32] derives environment models from user-provided environment assumptions which capture ordering relationships among program actions. OCSEGen [31] is an environment generation tool for open components and systems, which generates both drivers and environment stubs by analyzing Java byte-code. Modgen [12] makes uses of program slicing to generate an abstract model of a class, and it focuses on optimization of library classes by reducing their complexity. [33] discusses these two mentioned tools on how they can be applied to generate Android environment models. [23] acquires the input-output specification by sampling given binary implementations of the methods being modeled, and uses SMT solver to construct models which satisfy the specification. Unlike previous approaches, we automatically construct environment models by analyzing Javadocs which give abundant information about the behaviors of each library method.

7. CONCLUSION

Based on the idea of applying natural language processing techniques and program analysis techniques to model libraries, we identify and overcome the challenges of applying NLP techniques on real code generation, and build a prototype of a modeling tool that can automatically generate simplified code for Java container-like libraries from Javadocs. On average, the size of the generated model is only one third of that of the original code base. We also apply our technique to help other program analysis that needs reasoning software runtime environment. The results show that our models can help both static and dynamic analysis.

8. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their constructive comments. This research was partially supported by NSF grants CCF-1320326, CCF-0845870, CCF-0953759 and CCF-1319705, and the CAS/SAFEA international Partnership Program for Creative Research Teams, NSERC and an Ontario Early Researcher Award, National Key Basic Research Program of China No.2014CB340703, National Science Foundation of China No.61561146394, No.91318301, No.91418204, No.61321491 and No.61472179. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

9. REFERENCES

[1] Dacapo. http://dacapobench.org/.

[2] Funkyjfilter benchmark. https://github.com/olim7t/java-benchmarks/blob/master/src/main.java/FunkyJFilterBenchmark.java.
[3] J2SE's javadoc. http://docs.oracle.com/javase/8/docs/api/.
[4] Javaslicer. https://github.com/hammacher/javaslicer.
[5] Jtreg. http://openjdk.java.net/jtreg/.
[6] Listappend benchmark. https://github.com/olim7t/java-benchmarks/blob/master/src/main.java/ListAppendBenchmark.java.
[7] Specjbb. https://www.spec.org/.
[8] The stanford natural language processing group. http://nlp.stanford.edu/software/lex-parser.shtml, 1999.
[9] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Notices*. ACM, 1990.
[10] W. A. Andrew and P. Jens. Modern compiler implementation in java, 2002.
[11] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
[12] M. Ceccarello and O. Tkachuk. Automated generation of model classes for java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.
[13] D. Cristian Cadar and D. Dunbar. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI, San Diego, CA, USA (December 2008)*, 2008.
[14] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for java container classes. *Software Engineering, IEEE Transactions on*, 33(8):526–543, 2007.
[15] D. Jurafsky and J. H. Martin. Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition.
[16] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003.
[17] C. D. Manning and H. Schiitze. Foundations of statistical natural language processing.
[18] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.
[19] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.
[20] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84. IEEE, 2007.
[21] V. K. Palepu, G. Xu, J. Jones, et al. Improving efficiency of dynamic analysis with dynamic dependence summaries. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 59–69. IEEE, 2013.
[22] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language api descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, pages 815–825. IEEE Press, 2012.
[23] D. Qi, W. N. Sumner, F. Qin, M. Zheng, X. Zhang, and A. Roychoudhury. Modeling software execution environment. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 415–424. IEEE, 2012.
[24] A. N. Rafferty and C. D. Manning. Parsing three german treebanks: Lexicalized and unlexicalized baselines. In *Proceedings of the Workshop on Parsing German*, pages 40–46. Association for Computational Linguistics, 2008.
[25] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 499–510. IEEE, 2007.
[26] A. Sinha, S. M. Sutton, and A. Paradkar. Text2test: Automated inspection of natural language use cases. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 155–164. IEEE, 2010.
[27] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information systems security*, pages 1–25. Springer, 2008.
[28] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 45–54. ACM, 2010.
[29] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* icomment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158. ACM, 2007.
[30] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 260–269. IEEE, 2012.
[31] O. Tkachuk. Ocsegen: Open components and systems environment generator. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis*, pages 9–12. ACM, 2013.
[32] O. Tkachuk, M. B. Dwyer, and C. S. Păsăreanu. Automated environment generation for software model checking. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 116–127. IEEE, 2003.
[33] H. van der Merwe, O. Tkachuk, B. van der Merwe, and W. Visser. Generation of library models for verification of android applications. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, 2015.
[34] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
[35] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data

transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[36] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.

[37] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318. IEEE Computer Society, 2009.