Generating Method Documentation Using Concrete Values from Executions*

Matúš Sulír[1] and Jaroslav Porubän[2]

1 **Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Košice, Slovakia**
matus.sulir@tuke.sk

2 **Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Košice, Slovakia**
jaroslav.poruban@tuke.sk

Abstract

There exist multiple automated approaches of source code documentation generation. They often describe methods in abstract terms, using the words contained in the static source code or code excerpts from repositories. In this paper, we introduce DynamiDoc – a simple yet effective automated documentation approach based on dynamic analysis. It traces the program being executed and records string representations of concrete argument values, a return value, and a target object state before and after each method execution. Then for every concerned method, it generates documentation sentences containing examples, such as "When called on [3, 1.2] with element = 3, the object changed to [1.2]". A qualitative evaluation is performed, listing advantages and shortcomings of the approach.

1998 ACM Subject Classification D.2.7 [Distribution, Maintenance, and Enhancement] Documentation, D.2.5 [Testing and Debugging] Tracing, D.3.3 [Language Constructs and Features] Procedures, Functions, and Subroutines

Keywords and phrases documentation generation, source code summarization, methods, dynamic analysis, examples

Digital Object Identifier 10.4230/OASIcs.SLATE.2017.3

1 Introduction

When developers try to comprehend what a particular method or procedure in source code does or what are its inputs and outputs, they often turn to documentation. For instance, in Java, the methods can be documented by comments specially formatted according to the Javadoc specification [6]. Similar standards and possibilities exist in other languages.

However, the API (application programming interface) documentation is often incomplete, ambiguous, obsolete, lacking good examples, inconsistent or incorrect [23]. In the worst case, it is not present at all.

Consider the simplified excerpt from Java 8 API presented in Listing 1[1] – a method `getAuhority()` in the class `URL` and its documentation.

For a person who is not a domain expert in the field of Internet protocols, this documentation in certainly not as useful as it should be. To get at least partial understanding of what

* This work was supported by project KEGA 047TUKE-4/2016 Integrating software processes into the teaching of programming.

[1] `https://docs.oracle.com/javase/8/docs/api/java/net/URL.html#getAuthority--`

Listing 1 Excerpt from JAVA 8 API documentation.

```
public class URL {
  ...
  /**
    * Gets the authority part of this URL.
    * @return the authority part of this URL
    */
  public String getAuthority() {...}
}
```

the method does, he must study the corresponding RFC (Request for Comments) documents, various web tutorials and resources, or browse the rest of the lengthy API documentation.

Now, consider the same method, but with a documentation containing concrete examples of arguments and return values, as presented in Listing 2.

Listing 2 Excerpt from JAVA 8 API documentation enriched with concrete examples.

```
public class URL {
  ...
  /**
    * Gets the authority part of this URL.
    * @return the authority part of this URL
    * @examples When called on http://example.com/path?query,
    *     the method returned "example.com".<br>
    *     When called on http://user:password@example.com:80/path,
    *     the method returned "user:password@example.com:80".
    */
  public String getAuthority() {...}
}
```

This kind of documentation should give a developer instant "feeling" of what the method does and help him to understand the source code.

Creating and maintaining documentation manually is time-consuming and error-prone. There exist multiple approaches for automated documentation generation [13]. Many of these approaches work by combining static analysis with Natural Language Processing (NLP) [17, 10] or repository mining [5]. NLP-based and static analysis approaches have an inherent disadvantage: they only present information already available in the static source code in another way. Mining-based methods require large repositories of code using concerned APIs. Furthermore, none of the mentioned approaches provide concrete, literate string representations of arguments, return values and states from runtime: at best, they provide code examples which use the API. While there exist dynamic analysis approaches collecting run-time values of variables [8, 7], they are not oriented toward textual documentation generation.

In this paper, we describe a method documentation approach based on dynamic analysis. The program of interest is executed either manually or using automated tests. During these executions, a tracer saves string representations (obtained by calling a `toString()`-like method) of the methods' arguments, return value, and object states before and after executing the method. For each method, a few sample executions are chosen, and documentation sentences

Listing 3 The tracing algorithm executed around each method.

```
function around(method)
  // save string representations of arguments and object state
  arguments ← []
  for arg in method.args
    arguments.add(to_string(arg))
  end for
  before ← to_string(method.this)

  // run the original method, save return value or thrown exception
  result ← method(method.args)
  if result is return value
    returned ← to_string(result)
  else if result is thrown exception
    exception ← to_string(result)
  end if

  // save object representation again and write record to trace file
  after ← to_string(method.this)
  write_record(method, arguments, before, returned, exception, after)

  // proceed as usual (not affecting the program's semantics)
  return/throw result
end function
```

similar to the exhibit shown above are generated. The generated Javadoc documentation is then written into the source files.

To show the feasibility of our approach, named DynamiDoc, a prototype implementation was constructed.[2] We applied DynamiDoc on multiple open-source projects and performed qualitative evaluation: we will describe its current benefits and drawbacks.

2 Documentation approach

Now we will describe the documentation approach in more detail. Our method consists of three consecutive phases: tracing, selection of examples, and documentation generation.

2.1 Tracing

First, all methods in the project we want to document are instrumented to enable tracing. Each method's definition is essentially replaced by the code presented in Listing 3. In our implementation, we used bytecode-based AspectJ instrumentation; however, the approach is not limited to it.

The target project is then executed as usual. This can range from manual clicking in a graphical user interface of an application to running fully automatized unit tests of a library. Thanks to the mentioned instrumentation, selected information about each method execution is recorded into a trace file. A detailed explanation of the tracing process for one method execution follows.

[2] It is available at `https://github.com/sulir/dynamidoc`.

All parameter values are converted to their string representations (lines 3–6 in Listing 3). By a string representation, we mean a result of calling the `toString()` method[3] on an argument. For simple numeric and string types, it is straightforward (e.g., the number 7.1 is represented as `7.1`). For more complicated objects, it is possible to override the `toString()` method of a given class to meaningfully represent the object's state. For instance, `Map` objects are represented as `{key1=value1, key2=value2}`.

Each non-static method is called on a specific object (called `this` in Java), which we will call a "target object". We save the target object's string representation before actual method execution (line 7). This should represent the original object state. In our example from Section 1, a string representation of a URL object constructed using `new URL("http://example.com/path?query");` looks like `http://example.com/path?query`.

We execute the given method and convert the result to a string (lines 10–15). For non-void methods, this is the return value. In case a method throws an exception, we record it and convert to string – even exceptions has their `toString()` method. In the example we are describing, the method returned `example.com`.

After the method completion, we again save the string representation of the target object (`this`, line 18) if the method is non-static. This time, it should represent the state affected by the method execution. Since the `getAuthority()` method does not mutate the state of a URL object, it is the same as before calling the method.

Finally, we write the method location (the file and line number) along with the collected string representations to the trace file. We return the stored return value or throw the captured exception, so the program execution continues as it would do without our tracing code.

To sum up, a trace is a collection of stored executions, where each method execution is a tuple consisting of:

- *method* – the method identifier (file, line number),
- *arguments* – an array of string representations of all argument values,
- *before* – a string representation of the target object state before method execution,
- *return* – a string representation of the return value, if the method is non-void and did not throw and exception,
- *exception* – a thrown exception converted to a string (if it was thrown),
- *after* – a string representation of the target object state after method execution.

2.2 Selection of examples

After tracing finishes, the documentation generator reads the written trace file which contains a list of all method executions. Since one method may have thousands of executions, we need to select a few most suitable ones – executions which will be used as examples for documentation generation. Each execution is assigned a metric representing its suitability to be presented as an example to a programmer. They are then sorted in descending order according to this metric and the first few of them are selected. In the current implementation, we limit the number of examples for each method to 5.

In the current version of DynamiDoc, we use a very simple metric: execution frequency. It is a number of times which the method was executed in the same state with the same arguments and return value (or thrown exception) and resulted in the same final state –

[3] There are some exceptions – for example, on arrays, we call `Arrays.deepToString(arr)`. Similar methods exist in other languages, such as C# or Ruby.

considering the stored string representations. This means we consider the most frequent executions the most representative and use them for documentation generation. For instance, if the `getAuthority()` method was called three times on `http://example.com/path?query` producing `"example.com"`, and two times on `http://user:password@example.com:80/path` producing `"user:password@example.com:80"`, these two examples are selected, in the given order.

2.3 Documentation generation

After obtaining a list of a few example executions for each method, a documentation sentence is generated for each such execution. The generation process is template-based.

First, an appropriate sentence template is selected, based on the properties of the method (static vs. non-static, parameter count and return type) and the execution (whether an exception was thrown, or a string representation of the target object changed by calling the method). The selection is performed using a decision table displayed in Table 1. For instance, the `getAuthority()` method is non-static (instance), it has 0 parameters, does not have a void type, its execution did not throw an exception and the URL's string representation is the same before and after calling it (the state did not change from our point of view). Therefore, the sentence template "When called on {before}, the method returned {return}." is selected.

Next, the placeholders (enclosed in braces) in the sentence template are replaced by actual values. The meaning of individual values was described at the end of Section 2.1. An example of a generated sentence is: "When called on http://example.com/path?query, the method returned "example.com"." Note that we use past tense since in general, we are not sure the method always behaves the same way – the sentences represent some concrete recorded executions.

Finally, we write the sentences into Javadoc documentation comments of affected methods. Javadoc documentation is structured – it usually contains tags such as `@param` for a description of a parameter and `@see` for a link to a related class or method. We append our new, custom `@examples` tag with the generated documentation sentences to existing documentation. If the method is not yet documented at all, we create a new Javadoc comment for it. When existing examples are present, they are replaced by the new ones. The original source code files are overwritten to include the modified documentation. Using a custom "doclet" [6], the `@examples` tag can be later rendered as the text "Examples:" in the HTML version of the documentation.

3 Evaluation

Since the approach and its implementation is preliminary, we did not perform quantitative analysis of computational performance or program comprehension efficiency improvement. Instead, a qualitative evaluation was performed. We applied our documentation approach on three real-world open source projects and observed its strengths and weaknesses by inspecting the generated documentation.

The mentioned open source projects are:

- Apache Commons Lang,[4]
- Google Guava,[5]
- and Apache FOP.[6]

[4] `https://commons.apache.org/lang/`
[5] `https://github.com/google/guava`
[6] `https://xmlgraphics.apache.org/fop/`

Table 1 A decision table for the documentation sentence templates.

Method kind	Parameters	Return type	Exception	State changed	Sentence template
static	0	void	no	no	-
		non-void			The method returned {return}.
		any	yes		The method threw {exception}.
	≥ 1	void	no		The method was called with {arguments}.
		non-void			When {arguments}, the method returned {return}.
		any	yes		When {arguments}, the method threw {exception}.
instance	0	void	no	no	The method was called on {before}.
				yes	When called on {before}, the object changed to {after}.
		non-void		no	When called on {before}, the method returned {return}.
				yes	When called on {before}, the object changed to {after} and the method returned {return}.
		any	yes	no	When called on {before}, the method threw {exception}.
				yes	When called on {before}, the object changed to {after} and the method threw {exception}.
	≥ 1	void	no	no	The method was called on {before} with {arguments}.
				yes	When called on {before} with {arguments}, the object changed to {after}.
		non-void		no	When called on {before} with {arguments}, the method returned {return}.
				yes	When called on {before} with {arguments}, the object changed to {after} and the method returned {return}.
		any	yes	no	When called on {before} with {arguments}, the method threw {exception}.
				yes	When called on {before} with {arguments}, the object changed to {after} and the method threw {exception}.

The first two projects are utility libraries aiming to provide core functionality missing in the standard Java API. To obtain data for dynamic analysis, we executed selected unit tests of the libraries. The last project is a Java application reading XML files containing formatting objects (FO) and writing files suitable for printing, such as PDFs. In this case, we executed the application using a sample FO file as its input.

A description of selected kinds of situations we encountered and observations we made follows.

3.1 Utility methods

For simple static methods accepting and returning primitive or string values, our approach generally produces satisfactory results. As one of many examples, we can mention the method `static String unicodeEscaped(char ch)` in the "utility class" `CharUtils` of

Apache Commons Lang. The generated sentences are in the form:

```
When ch = 'A', the method returned "\u0041".
```

For many methods, the Commons Lang API documentation already contains source code or pseudo-code examples. Here is an excerpt from the documentation of the aforementioned method:

```
CharUtils.unicodeEscaped('A') = "\u0041"
```

Even in cases when a library already contains manually written code examples, DynamiDoc is useful for utility methods on simple types:

- to save time spent writing examples,
- to ensure the documentation is correct and up-to-date.

The latter point is fulfilled when the tool is run automatically, e.g., as a part of a build process. Sufficient unit test coverage is a precondition for both points.

3.2 Data structures

Consider the data structure `HashBasedTable` from Google Guava and its method `size()` implemented in the superclass `StandardTable`. The DynamiDoc-generated documentation includes this sentence:

```
When called on {foo={1=a, 3=c}, bar={1=b}}, the method returned 3.
```

Compare it with a hypothetical manually constructed source-code based example:

```
Table<String, Integer, Character> table = HashBasedTable.create();
table.put("foo", 1, 'a');
table.put("foo", 3, 'c');
table.put("bar", 1, 'b');
System.out.println(table.size()); // prints 3
```

Instead of showing the whole process how we got to the given state, our approach displays only a string representation of the object state (`{foo={1=a, 3=c}, bar={1=b}}`). Such a form is very compact and still contains sufficient information necessary to comprehend the gist of a particular method.

3.3 Changing target object state

When the class of interest has the method `toString()` meaningfully overwritten, DynamiDoc works properly. For instance, see one of the generated documentation sentences of the method `void FontFamilyProperty.addProperty(Property prop)` in Apache FOP:

```
When called on [sans-serif] with prop = Symbol, the object changed to
[sans-serif, Symbol].
```

Now, let us describe an opposite extreme. In the case of Java, the default implementation of the `toString()` method is not very useful: it displays just the class name and the object's hash code. When the class of interest does not have the `toString()` method overridden, DynamiDoc does not produce documentation of sufficient quality. Take, for example, the generated documentation for the method `void LayoutManagerMapping.initialize()` in the same project:

```
The method was called on
org.apache.fop.layoutmgr.LayoutManagerMapping@260a3a5e.
```

While the method probably changed the state of the object, we cannot see the state before and after calling it. The string representation of the object stayed the same – and not very meaningful.

Although this behavior is a result of an inherent property of our approach, there exists a way how this situation can be improved: to override `toString()` methods for all classes when it can be at least partially useful. Fortunately, many contemporary IDEs support automated generation of `toString()` source code. Such generated implementations are not always perfect, but certainly better than nothing.

We plan to perform an empirical study assessing what portion of existing classes in open source projects meaningfully override the `toString()` method. This will help us to quantitatively assess the usefulness of DynamiDoc.

3.4 Changing argument state

The current version of DynamiDoc does not track changes of the passed parameter values. For example, the method `static void ArrayUtils.reverse(int[] array)` in Apache Commons Lang modifies the given array in-place, which is not visible in the generated documentation:

```
The method was called with array = [1, 2, 3].
```

Of course, it is possible to compare string representations of all mutable objects passed as arguments before and after execution. We can add such a feature to DynamiDoc in the future.

3.5 Operations affecting external world

Our approach does not recognize the effects of input and output operations. When such an operation is not essential for the method, i.e., it is just a cross-cutting concern like logging, it does not affect the usefulness of DynamiDoc too much. This is, for instance, the case of the method `static int FixedLength.convert(double dvalue, String unit, float res)` in Apache FOP. It converts the given length to millipoints, but also contains code which logs an error when it occurs (e.g., to a console). A sample generated sentence follows:

```
When dvalue = 20.0, unit = "pt" and res = 1.0, the method returned
20000.
```

On the other hand, DynamiDoc is not able to generate any documentation sentence for the method `static void CommandLineOptions.printVersion()`, which prints the version of Apache FOP to standard output.

3.6 Methods doing too much

Our approach describes methods in terms of their overall effect. It does not analyze individual actions performed during method execution. Therefore, it is difficult to generate meaningful documentation for methods such as application initializers, event broadcasters or processors. An example is the method `void FObj.processNode(String elementName, Locator locator, Attributes attlist, PropertyList pList)`. An abridged excerpt from the generated documentation follows.

```
The method was called on ...RegionAfter@206be60b[@id=null] with
elementName = "region-after", locator = ...LocatorProxy@292158f8,
attlist = ...AttributesProxy@4674d90
and pList = ...StaticPropertyList@6354dd57.
```

3.7 Example selection

The documentation of some methods is not the best possible one. For instance, the examples generated for the method `BoundType Range.lowerBoundType()` in Google Guava are:

```
When called on (5..+∞), the method returned OPEN.
When called on [4..4], the method returned CLOSED.
When called on [4..4), the method returned CLOSED.
When called on [5..7], the method returned CLOSED.
When called on [5..8), the method returned CLOSED.
```

This selection is not optimal. First, there is only one example of the OPEN bound type – but this is only a cosmetic issue. The second, worse flaw is the absence of a case when the method throws an exception (`IllegalStateException` when the lower bound is $-\infty$).

The method `Range encloseAll(Iterable values)` in the same class has much better documentation, which shows the variety of inputs and outputs (although ordering could be slightly better):

```
When values = [0], the method returned [0..0].
When values = [5, -3], the method returned [-3..5].
When values = [0, null], the method threw
    java.lang.NullPointerException.
When values = [1, 2, 2, 2, 5, -3, 0, -1], the method returned [-3..5].
When values = [], the method threw java.util.NoSuchElementException.
```

Improvement of the example selection metric will be necessary in the future. A possible option is to include the most diverse examples: some short values, some long, plus a few exceptions.

Furthermore, like in any dynamic analysis approach, care must be taken not to include sensitive information like passwords in the generated documentation.

4 Related work

In this section, we will present related work with a focus on documentation generation and source code summarization. The related approaches are divided according to the primary analysis type they use – static analysis (sometimes enhanced by repository mining) or dynamic analysis.

4.1 Static analysis and repository mining

Sridhara et al. [16, 17] generate natural-language descriptions of methods. The generated sentences are obtained by analyzing the words in the source code of methods; therefore, it does not contain examples of concrete variable values which can be often obtained only at runtime. In [18], they add support for parameter descriptions, again using static analysis only.

McBurney and McMillan [10] summarize also method context – how the method interacts with other ones. While the approach considers source code outside the method being described, they still use only static analysis.

Buse and Weimer [1] construct natural language descriptions of situations when exceptions may be thrown. These descriptions are generated for methods, using static analysis.

Long et al. [9] describe an approach which finds API functions most related to the given C function. If adapted to Java, it could complement DynamiDoc's documentation by adding `@see` tags to Javadoc.

Moreno el al. [12] automatically document classes instead of methods. Their natural language descriptions utilize class stereotypes like "Entity", determined using source code analysis.

The tool eXoaDocs [5] mines large source code repositories to find usages of particular API elements. Source code examples illustrating calls to API methods are then added to generated Javadoc documentation. In Section 3.2, we described the difference between source code based examples and examples based on string representations of concrete variable values. Furthermore, the main point of source code examples is to show how to use the given API. Therefore, the descriptions of results or outputs are often not present in the examples.

Buse and Weimer [2] synthesize API usage examples. Compared to Kim et al. [5], they do not extract existing examples from code repositories, but use a corpus of existing programs to construct new examples.

The APIMiner platform [11] uses a private source code repository to mine examples. The generated Javadoc documentation contains "Examples" buttons showing short code samples and related elements.

4.2 Dynamic analysis

Hoffman and Strooper [4] present an approach of executable documentation. Instead of writing unit tests in separate files, special markers in method comments are used to mark tests. Each test includes the code to be executed and an expected value of the expression, which serves as specification and documentation. Compared to our DynamiDoc, they did not utilize string representations of objects – the expected values are Java source code expressions. Furthermore, in the case of DynamiDoc, the run-time values can be collected from normal program executions, not only from unit tests.

Concern annotations [19] are Java annotations above program elements such as methods, representing the intent behind the given piece of code. AutoAnnot [20] writes annotations representing features (e.g., `@NoteAdding`) above methods unique to a given feature obtained using simple dynamic analysis. While AutoAnnot describes methods using only simple identifiers, DynamiDoc generates full natural language sentences containing concrete variable values.

ADABU [3] uses dynamic analysis to mine object behavior models. It constructs state machines describing object states and transitions between them. Compared to our approach which used class-specific string representations, in ADABU, an object state is described using a predicate like "isEmpty()". Furthermore, they do not provide examples of concrete parameter and return values.

Lo and Maoz [8] introduce a combination of scenario-based and value-based specification mining. Using dynamic analysis, they generate live sequence charts in UML (Unified Modeling Language). These charts are enriched with preconditions and postconditions containing string representations of concrete variable values. However, their approach does not focus on

documentation of a single method, its inputs and outputs; rather they describe scenarios of interaction of multiple cooperating methods and classes.

Tralfamadore [7] is a system for analysis of large execution traces. It can display the most frequently occurring values of a specific function parameter. However, it does not provide a mapping between parameters and a return value. Furthermore, since it is C-based, it does not present string representations of structured objects.

TestDescriber by Panichella et al. [14] executes automatically generated unit tests and generates natural language sentences describing these tests. First, it differs from DynamiDoc since their approach describes only the unit tests themselves, not the tested program. Second, although it uses dynamic analysis, the only dynamically captured information by TestDescriber is code coverage – they do not provide any concrete variable values except that present literally in the source code.

FailureDoc [25] observes failing unit test execution. It adds comments above individual lines inside tests, explaining how the line should be changed for the test to pass.

SpyREST [15] generates documentation of REST (representational state transfer) services. The documentation is obtained by running code examples, intercepting the communication, and generating concrete request-response examples. While SpyREST is limited to web applications utilizing the REST architecture, DynamiDoc produces documentation of any Java program.

@tComment [22] is a tool using dynamic analysis to find code-comment inconsistencies. Unlike DynamiDoc, it does not produce new documentation – it only checks existing, manually written documentation for broken rules.

5 Conclusion and future work

In this paper, we presented DynamiDoc – a novel approach of automated method documentation and its preliminary implementation. It traces program execution to obtain concrete examples of arguments, return values, and object states before and after calling a method. Thanks to `toString()` methods, the values are converted to strings during the program runtime and only these converted values are stored in a trace. For each method, a few execution examples are selected, and natural-language sentences are generated and integrated into Javadoc comments.

The approach should facilitate program comprehension. The documentation generated by DynamiDoc is not intended as a complete replacement for manually written documentation, but it can be a good complement. Compared to natural language processing and simple static analysis, automated documentation approaches utilizing dynamic analysis require more effort, e.g., periodically building the software (which often fails [21]) and executing automated tests or running the software manually (which takes time). An investigation whether this additional effort is worth the benefits provided by the generated documentation should be performed.

The currently presented version has some shortcomings which we would like to mitigate before assessing its effect on code understanding.

First, the string representations of objects are not always ideal. We plan to find out the current prevalence of `toString()` methods in Java (and probably C# and other) classes, determine the "state of the art" in object string representation generation and try to improve it.

Second, we do not have empirical findings what are the attributes of a "useful example". Therefore, we used only a very simple metric of execution frequency to sort the example

executions and select the best ones. We would like to investigate, both qualitatively and quantitatively, what examples are considered the best by developers, and modify the selection metric accordingly. Existing knowledge in the area of test prioritization [24] and source code example selection [5] can be adapted and extended.

Finally, we could record also the changes of argument states and interactions with external world (input/output operations) to improve the generated summaries. In cases when summaries of overall effects of methods are insufficient, describing also individual actions inside them using runtime values of variables could help.

References

1. Raymond Buse and Westley Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis*, pages 273–282. ACM, 2008.
2. Raymond Buse and Westley Weimer. Synthesizing API usage examples. In *34th International Conference on Software Engineering (ICSE'2012)*, pages 782–792, 2012.
3. Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *International Workshop on Dynamic Systems Analysis*, pages 17–24. ACM, 2006.
4. Daniel Hoffman and Paul Strooper. Prose + Test Cases = Specifications. In *34th International Conference on Technology of Object-Oriented Languages and Systems*, pages 239–250, 2000.
5. Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Adding examples into Java documents. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 540–544, 2009.
6. Douglas Kramer. API documentation from source code comments: A case study of Javadoc. In *17th Annual International Conference on Computer Documentation*, pages 147–153, 1999.
7. Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. Execution mining. In *8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, pages 145–158, 2012.
8. David Lo and Shahar Maoz. Scenario-based and value-based specification mining: Better together. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 387–396, 2010.
9. Fan Long, Xi Wang, and Yang Cai. API hyperlinking via structural overlap. In *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 203–212, 2009.
10. Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *22nd International Conference on Program Comprehension*, pages 279–290, 2014.
11. João E. Montandon, Hudson Borges, Daniel Felix, and Marco T. Valente. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *20th Working Conference on Reverse Engineering (WCRE'2013)*, pages 401–408, October 2013.
12. Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for Java classes. In *21st International Conference on Program Comprehension*, pages 23–32, May 2013.
13. Najam Nazar, Yan Hu, and He Jiang. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, 31(5):883–909, 2016.
14. Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *38th International Conference on Software Engineering*, pages 547–558, 2016.

15 Sheikh Mohammed Sohan, Craig Anslow, and Frank Maurer. SpyREST: Automated RESTful API documentation using an HTTP proxy server. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 271–276, 2015.

16 Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52, 2010.

17 Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *33rd International Conference on Software Engineering*, pages 101–110, 2011.

18 Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *19th IEEE International Conference on Program Comprehension*, pages 71–80, 2011.

19 Matúš Sulír, Milan Nosáľ, and Jaroslav Porubän. Recording concerns in source code using annotations. *Computer Languages, Systems & Structures*, 46:44–65, November 2016.

20 Matúš Sulír and Jaroslav Porubän. Semi-automatic concern annotation using differential code coverage. In *IEEE 13th International Scientific Conference on Informatics*, pages 258–262, November 2015.

21 Matúš Sulír and Jaroslav Porubän. A quantitative study of Java software buildability. In *7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–25, 2016.

22 Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Fifth IEEE International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012.

23 Gias Uddin and Martin P. Robillard. How API documentation fails. *IEEE Software*, 32(4):68–75, July 2015.

24 Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

25 Sai Zhang, Cheng Zhang, and Michael D. Ernst. Automated documentation inference to explain failed tests. In *26th IEEE/ACM International Conference on Automated Software Engineering*, pages 63–72, 2011.