Synthesizing API Usage Examples

Raymond P.L. Buse and Westley Weimer
Department of Computer Science
University of Virginia, Charlottesville, VA, USA
{*buse,weimer*}*@cs.virginia.edu*

***Abstract*—Key program interfaces are sometimes documented with usage examples: concrete code snippets that characterize common use cases for a particular data type. While such documentation is known to be of great utility, it is burdensome to create and can be incomplete, out of date, or not representative of actual practice.**

We present an automatic technique for mining and synthesizing succinct and representative human-readable documentation of program interfaces. Our algorithm is based on a combination of path sensitive dataflow analysis, clustering, and pattern abstraction. It produces output in the form of well-typed program snippets which document initialization, method calls, assignments, looping constructs, and exception handling. In a human study involving over 150 participants, 82% of our generated examples were found to be at least as good at human-written instances and 94% were strictly preferred to state of the art code search.

I. Introduction

Professional software developers spend most of their time trying to understand code [1], [2]. Maintaining and evolving high-quality documentation is crucial to help developers understand and modify code [3], [4]. In reports by and studies of developers, API use examples have been found to be a key learning resource [5], [6], [7], [8], [9], [10]. That is, documenting how to *use* an API is preferable to simply documenting the function of each of its components.

One study found that the greatest obstacle to learning an API in practice is "insufficient or inadequate examples." [11] We present an algorithm that automatically generates API usage examples. Given a data-type and software corpus (i.e., a library of programs that make use of the data-type), we extract abstract use-models for the data-type and render them in a form suitable for use by humans as documentation.

The state of the art in automated support for usage examples in known as *code search*. Typically, the problem is phrased as one of ranking concrete code snippets on criteria such as "representativeness" and "conciseness." In 2009, Zhong *et al.* described a technique called MAPO for mining and recommending example code snippets [12]. More recently, Kim *et al.* presented a tool called EXoaDocs which also finds and ranks code examples for the purpose of supplementing JavaDoc embedded examples [13]. Such examples can be useful, but they are very different from human-written examples. Mined examples often contain extraneous statements, even when slicing is employed. In addition, they often lack the context required to explicate the material they present. In general, mined examples are long, complex, and difficult to understand and use. Good human-written examples, on the other hand, often present only the information needed to understand the API and are free of superfluous context. Human written documentation has two important disadvantages, however: it requires a significant human effort to create, and is thus often not created; and it may not be representative of, or up-to-date with, actual use.

In this paper we present a technique for automatically synthesizing human-readable API usage examples which are well-typed and representative. We adapt techniques from specification mining [14] to model API uses as graphs describing method call sequences, annotated with control flow information. We use data-flow analysis to extract important details about each use beyond the sequence of method calls, such as how the type was initialized and how return values are used. We then abstract concrete uses into high-level examples. Because a single data-type may have multiple common use scenarios, we use clustering to discover and coalesce related usage patterns before expressing them as documentation.

Our generated examples display a number of important advantages over both state-of-the-art code search and human written examples, both of which we compare to in a human study. Unlike mined examples, our generated examples contain only the program statements needed to demonstrate the target behavior. Where concrete examples can be needlessly specific, our examples adopt the most common types and names for identifiers. Unlike human-written examples, our examples are, by construction, well-formed syntactically and well-typed. Where previous approaches to code ranking adopted simple heuristics based on length and a simple use count (e.g., [15], [16]), our abstract examples are structured and generated with a robust and well-defined notion of representativeness. Because our approach is fully automatic, the examples are also cheap to construct and can be always up-to-date. Additionally, their well-formedness properties make them ideal automated tasks like for code completion [17].

The main contributions of this paper are:

- A study of API usage examples. We characterize gold-standard human-written examples from the Java SDK. Furthermore, we present results from a large survey on example quality and utility.
- An algorithm for the automatic construction of example

documentation. The algorithm takes as input a software corpus and a target data-type. It produces a ranked list of well-typed, human-readable code snippets which exemplify typical use of that data-type.

- A prototype implementation of the algorithm, and a comparison of its output to human-written example documentations. A human study with over 150 participants suggests our tool could replace as many as 82% of human-generated examples.

II. Motivating Example

In modern software development, API documentation tools such as JavaDoc have become increasingly prevalent, and variants exist for most languages (e.g., PythonDoc, OcamlDoc, etc.). One of the principles of JavaDoc is "including examples for developers" [18]. Not all examples are created equal, however. Features such as conciseness, representativeness, well-chosen variable names, correct control flow, and abstraction all relate to documentation quality.

Consider Java's `BufferedReader` class, which provides a buffering wrapper around a lower-level, non-buffered stream. The human-written usage example included in the official Java Development Kit, version 6 [19] is:

```
BufferedReader in =
 new BufferedReader(new FileReader("foo.in"));
```

While this example has the merit of being concise, it shows only how to create a `BufferedReader`, not how to use one. By contrast, our algorithm produces:

```
FileReader f; //initialized previously
BufferedReader br = new BufferedReader(f);
while(br.ready()) {
   String line = br.readLine();
   //do something with line
}
br.close();
```

This exemplifies one common usage pattern for a `BufferedReader`: repeatedly calling its `readLine` method while it remains `ready`. The variable names `br` and `f` were selected from among the most common human choices for `BufferedReader` and `FileReader`, and were synthesized together here: no single usage example need exist that uses both of those names in tandem. In addition, the example also demonstrates the importance of control flow: `readLine` is called repeatedly, but only after checking `ready`. Finally, the `//initialized previously` and `//do something with line` comments indicate points where different human developers would write different code and highlight the most direct places for a developer to adapt this code example into an existing setting.

In practice, there is more than one way to use a `BufferedReader`. Our algorithm can produce a ranked list of examples based on clusters of representative human usages. The second example we produce is:

```
InputStreamReader i; //initialized previously
BufferedReader reader = new BufferedReader(i);
String s;
while ((s = reader.readLine()) != null)
   //do something with s
}
reader.close();
```

This second example shows that other concrete argument types can be used to create a `BufferedReader` (e.g., a `InputStreamReader` can be used as well as a `FileReader`). In addition, it shows that there is a different usage pattern that involves always calling `readLine` but then checking the return value against `null` (rather than calling `ready`). Both of the examples produced by our algorithm are well-formed and introduce commonly-named, well-typed temporaries for function arguments and return values.

It is also possible to use code search and slicing techniques to produce API examples. Such a tool from Kim *et al.* [13] produces an output consisting of more than 14 lines on the same `BufferedReader` query (not shown).

Because they lack information related program semantics, slicing based approaches have trouble distinguishing between relevant an irrelevant details in an example. In the output of Kim *et al.*'s tool, a `BufferedReader` is initialized with `System.in`. To a new user, it may be difficult to tell if this argument is necessary, or as in this case, coincidental. Variable names are also often too specific to the example: `String acl_in = br.readLine()` (Kim *et al.*'s tool) is less descriptive than `String line = br. readLine()` for the general case. Furthermore, sliced examples do not type-check out of context and include many irrelevant statements.

We seek to produce high-quality usage example documentation automatically. To do so we must first understand how humans write usage examples and what they look for in usage documentation.

III. Human-Written Usage Examples

In this section we present a study of human-written API examples. The purpose of this study is to establish key guidelines for automatically creating examples that share the best properties of human-written ones. We seek to determine the desired size and readability of examples, and the importance of generality and correctness. We answer such questions by analyzing examples found in the standard Java API docs and through the results of a brief survey on the importance of various characteristics of examples.

A. Properties of Human-Written Examples

We explore the Java Software Development Kit (SDK) documentation because it is authoritative, generally considered to be of high quality, and is written for a general audience of Java developers. Furthermore, each example is

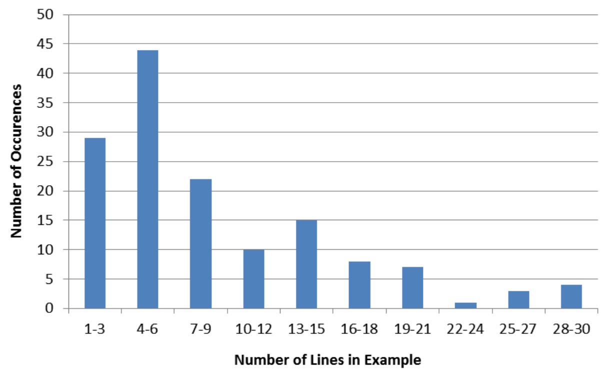

Figure 1. Histogram of the sizes (in lines) of usage examples embedded in the standard Java API documents.

tied to a specific class (and not, for example, to a coding task or other request).

We identified examples in the Java SDK by searching for pre-formatted text (e.g., contained in `<pre>` or `<code>` tags) and found 234 instances. We then manually inspected each one, selecting only those consisting of Java statements (e.g., ignoring lists of methods and other text). In all, we found 47 classes (3% of the total) which contain a usage example.

Length. We first consider the sizes of examples measured in lines of code. The full distribution of example sizes is presented in Figure 1. On average, human-written examples were 11 lines long, but the median size was 5 lines. While a significant number of examples are quite long, we note that human-written examples are typically very concise. We hypothesize that automated examples should be of a similar length to agree with human preferences and make a suitable supplement for existing examples.

Abstract initialization. We also note that many human-written examples use special markers, such as ellipses, to indicate that an input variable should be assigned a context-specific value. For example, the `glyphIndex` variable in this documentation for `java.awt.font.GlyphMetrics`:

```
int glyphIndex = ...;
GlyphMetrics metrics =
   GlyphVector.getGlyphMetrics(glyphIndex);
int isStandard = metrics.isStandard();
float glyphAdvance = metrics.getAdvance();
```

Similarly, an example from `text.MessageFormat` uses variables initialized with `7` and ``` ``a disturbance in the Force'' ```, which are clearly chosen arbitrarily.

```
int planet = 7;
String event = "a disturbance in the Force";
String result = MessageFormat.format(
   "At {1,time} on {1,date}, there was {2}" +
   "on planet {0,number,integer}.",
   planet, new Date(), event);
```

We hypothesize that automatically-constructed documentation should also use such abstract initialization to highlight "inputs" to the examples.

We hypothesize that generated documentation should employ common concrete type and variable names where applicable, but only when they represent truly frequent usage patterns. The algorithm presented in this paper annotates all variable which should be initialized with additional context with an `//initialized previously` comment.

Abstract use. Many examples contained an abstract placeholder indicating where the user should insert context-specific usage code. For example, the documentation for `java.text.CharacterIterator` makes it clear that the user should "do something with" or otherwise process the character `c` inside the loop:

```
for(char c = iter.first();
    c != CharacterIterator.DONE;
    c = iter.next()) {
  processChar(c);
}
```

We hypothesize that synthesized examples must similarly and concisely indicate where context-specific user code should be placed and what variables it should manipulate. The algorithm presented in this paper employs `//do something with X` annotations in such cases.

Exception Handling. 16 of the 47 examples contained some exception handling. Reasoning about programs that use exceptions is difficult for humans and also for automatic tools and analyses (e.g., [20], [21], [22]). Often exceptions are caught trivially (i.e., no action is taken to resolve the underlying error [23]) or the mechanism is purposely circumvented [24], [25]. This example for `java.nio.file.FileVisitor` is indicative:

```
try {
  file.delete();
} catch (IOException exc) {
  // failed to delete, do error handling here
}
return FileVisitResult.CONTINUE;
```

We hypothesize that tool-generated documentation should mention exception handling, but only when it is common or necessary for correctness. The algorithm presented in this paper learns common exception handling patterns in the corpus and distills examples representative of exception handling practice.

B. Survey Results

As part of the evaluation of the tool presented in this paper, we conducted a human study involving over 150 participants, primarily undergraduates at The University of Virginia enrolled in a class entitled *Software Development Methods* which places a heavy focus on learning the Java language and its standard set of APIs. Details about the study can be found in Section VII-A. As part of the study, humans were asked "What factors are important in a good example?" Some common themes emerged:

Multiple uses. Users wanted examples of different ways to use the class: "It shouldn't model something extremely specific, it should include something that the class will be most commonly used for." Documentation "must be able to show multiple uses. If you just show one example of a possible use, it probably won't be the one that interests me." We should "show examples of different ways a class can be used." The best documentation shows "all the different ways to use something, so it's helpful in all cases." Our algorithm uses clustering to capture multiple uses.

Readability. Users wanted examples that were easy to read and understand: "a good example is easy to understand and read." Many were explicit: "readable and understandable are the most important aspects." Slicing away unrelated code was critical: "less irrelevant, unrelated stuff in the example is better." A key component of readability is conciseness: the example should use "as little code as possible" and show "the most basic version of the problem. You can have additional more complicated problems if necessary." In Section VI-B we empirically demonstrate that our algorithm produces readable examples.

Naming. One key aspect of readability was the choice of identifier names: "they should be simple and understandable." Users preferred "logical variable names", "declaring variable names to represent what they do/are", "clear naming of variables", and variables demonstrating "standard naming". The importance of identifier names has been previously studied (e.g., [26]). Our algorithm tracks common variable naming information from concrete source code to produce understandable variable names.

Variables. Users also prefer documentation that includes intermediate variables and temporaries: "showing declarations" and "the use of many temporary variables helps" to make good documentation. The human-written documentation in Section II elides temporary variables; by contrast our algorithm produces well-typed, clearly-named temporaries.

The study also included a set of Likert-scale questions concerning the importance of several aspects of usage examples. The properties were *size* ("How concise is the example?"), *readability* ("How easy is it to understand?"), *representativeness* ("Will it be useful for what I want to do?"), and *concreteness* ("Can I compile and use it?").

Figure 2 shows the results. Notably, readability was the most important of the four features, with almost every participant ranking it as "very important" or "important." Note that readability is judged even more important than representativeness, which as been traditionally considered most important (e.g., [13]). Representativeness and size were the next most important, and concreteness was least important among the four. However, all four features were at least "important" on average, and must thus inform the design of our automatic example documentation algorithm.

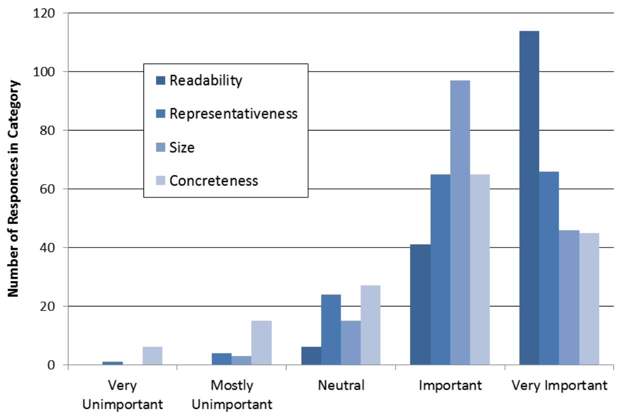

Figure 2. Human survey responses about the importance of various features of example documentation.

IV. Algorithm Description

This section describes our documentation-generation algorithm. The algorithm is designed to produce documentation with the qualities found in human-written examples (Section III-A) and the qualities praised by humans in our survey (Section III-B).

Figure 3 formalizes our algorithm in pseudo code and is referenced throughout this section. Our algorithm has four key phases. In *Path Enumeration* (Section IV-A) we statically enumerate intra-procedural traces. In *predicate generation* (Section IV-B), paths are merged into a smaller number of *concrete uses*. Conceptually, each concrete use corresponds a single (static) instantiation of the target type. The third step applies a *clustering* algorithm to identify groups of concrete uses that are similar (Section IV-C). These clusters are then merged into a small set of *abstract uses* which intuitively correspond to the ways the class is used in the corpus. Finally, we sort the abstract uses by their representativeness, flatten them into a "best method ordering" and distill them into human-readable documentation (Section IV-D).

A. Path Enumeration

For efficiency, we first filter the entire corpus, selecting only files that include a reference to the target class. We then process each method in each of the remaining classes of the corpus in turn. Note, however, that example generation does not require complete coverage of a corpus. Once a sufficiently large number of examples are found such that the model becomes stable the process can terminate.

We first enumerate the loop-free control flow paths of each method (Figure 3 lines 1–4): our analysis is thus path-sensitive and potentially exponential. We obtain loop-free paths by adding a statement to a path at most once: in effect, we consider each loop to either be taken once, or not at all. This decision can occasionally result in imprecise or incorrect documentation, however we see in Section VII that this occurs infrequently in practice.

```
Input: Target class $T$.
Input: Software corpus $Corp$.
Input: Distance metric on concrete uses $Dist$.
Input: Clustering parameter $k$.
Input: Least-upper-bound operator on statement set $\sqcup$.
Input: Comparison operator on statement lattice $\sqsubseteq$.
— Path Enumeration —
 1: let $paths \leftarrow \emptyset$
 2: for all method $m$ in $Corp$ do
 3:   if $m$ references $T$ then
 4:     $paths \leftarrow paths \cup$ EnumerateAcyclicPaths($m$)
— Symbexe & Use seeds —
 5: let $seeds \leftarrow \emptyset$
 6: let $symexe\_paths \leftarrow \emptyset$
 7: for all corpus paths $path$ in $paths$ do
 8:   let $symexe\_path \leftarrow [\ ]$
 9:   for all statements $stmt$ in $path$ in order do
10:     $\langle path\_preds, sym\_reg \rangle \leftarrow$ Symbexe($path, stmt$)
11:     for all $\langle p, q \rangle \in sym\_reg$ do
12:       $symexe\_stmt \leftarrow stmt[p \mapsto q]$
13:     $symexe\_path \leftarrow$
          $symexe\_path + \langle symexe\_stmt,\ path\_preds,\ stmt \rangle$
14:     for all sub-expressions $e$ in $stmt$ do
15:       if $e$ matches new $S$ $\vee$ ($e$ matches $obj.fld$ $\wedge$
          typeof($fld$) $=$ $T$) $\vee$ ($e$ is a function parameter
          $\wedge$ typeof($e$) $= T$) then
16:         $seeds \leftarrow seeds \cup \{e\}$
17:   $symexe\_paths \leftarrow symexe\_paths \cup \{symexe\_path\}$
— Concrete Uses —
18: let $concrete\_uses \leftarrow \emptyset$
19: for all expressions $seed$ in $seeds$ do
20:   let $seed\_paths \leftarrow \emptyset$
21:   for all paths $p$ in $symexe\_paths$ do
22:     let $sliced\_path \leftarrow$
          $\{s \mid s \in p \wedge seed$ is a sub-expression of $s\}$
23:     $seed\_paths \leftarrow seed\_paths \cup \{sliced\_path\}$
24:   let $nodes \leftarrow \{s \mid \exists p \in seed\_paths\ .\ s \in p\}$
25:   let $edge\_w(s_1, s_2) \leftarrow$
          $\mid \{p \mid p \in seed\_paths \wedge p =< \ldots s_1 \ldots s_2 \cdots >\} \mid$
26:   $concrete\_uses \leftarrow concrete\_uses \cup \{\langle nodes, edge\_w \rangle\}$
27: for all $c \in concrete\_uses$ do
28:   NormalizeEdgeWeights($c$)
— Abstract Uses —
29: let $clusters \leftarrow$ KMedoids($k, concrete\_uses, Dist$)
30: for all clusters $C \in clusters$ do
31:   let $nodes = \sqcup \{abst \mid \exists \langle N, E \rangle \in C\ .\ abst \in N\}$
32:   for all $\langle abst_1, abst_2 \rangle \in nodes$ do
33:     $abst\_edge\_w(abst_1, abst_2) \leftarrow 0$
34:     for all $\langle con_1, con_2 \rangle \in N\ .\ \langle N, E \rangle \in C$ do
35:       if $con_1 \sqsubseteq abst_1 \wedge con_2 \sqsubseteq abst_2$ then
36:         $abst\_edge\_w(abst_1, abst_2)$ += $E(con_1, con_2)$
37:   let $ordered \leftarrow \underset{s_1 \ldots s_n \in nodes\,!}{\operatorname{argmax}}\ Flow(s_1 \ldots s_n, abst\_edge\_w)$
38:   Output: GenerateCode($ordered$)
```

Figure 3. High-level pseudo code of our algorithm.

B. Predicate Generation

Next, we use symbolic execution [27] to compute intraprocedural *path predicates*, logical formulae that describe conditions under which each statement can be reached [28], [29] (Figure 3 line 10). To obtain these formulae, each control flow path is symbolically executed. We track conditional statements (e.g., `if` and `while`) and evaluate their guarding predicates using the current symbolic values for variables. We collect the resulting predicates; in conjunction, they form the path predicate for a given statement in the method.

In addition to the statement itself and its path predicates, we also record a version of the statement where each subexpression is replaced by the current value of that expression from the symbolic register file (lines 11–13). For example, `x.add(s)` might become `{new LinkedList()}.add("Hello")`.

Next we identify *use seeds* [14], expressions which represent a static instantiation of the target type: new object allocations, field references, and function parameters (lines 15–16). Intuitively, each such expression corresponds to one use of the class.

Seeds are the basis for the formation of *concrete uses* which join together all statements that are relevant to the object instantiation represented by the seed (lines 22–23). Concrete uses capture method ordering in a graph data structure where edges express the *happens before* relationship (cf. many specification mining techniques [30], [31], [32] where the edges represent method calls and an accepting sequence corresponds to a valid ordering) (lines 24–26). Employing class fields as seeds is of particular importance for Java, where method invocations on the same dynamically-allocated object often occur in multiple procedures. For example, a `HashTable` may be allocated in a constructor, populated in a second method and queried in a third. Although no *happens before* relation can be established between statements in separate methods in a purely intra-procedural analysis (except that constructors must always be called first), this permits the ordering between invocations located in the same method to be preserved. We have found this is critical to retaining sufficient ordering information to emit representative documentation.

C. Clustering and Abstraction

Even for a single target class, there are often many representative usage patterns to be found within a software corpus. We desire a ranked list of such patterns. We thus cluster the concrete uses from the previous step into *abstract uses*, abstractions of concrete uses where nodes are abstractions of multiple statements.

Intuitively, two concrete uses are similar and should be viewed as examples of the same abstract use pattern if they perform the same operations on the same data-types in the same order. We thus propose a formal *distance metric* between concrete uses which captures both statement ordering from the underlying state machines and type information. Our distance metric is similar to Kendall's τ [33], and captures both the number of sorting operations and number of type substitutions needed to transform one concrete use into another (see below). Our distance function is parametrized by α and β which express the relative importance of method ordering and type information. Our experience generating documentation for popular Java classes suggests that output quality is not highly sensitive to choice of α and β. However, in languages other than Java, particularly those with weak or dynamic type systems, selection of these parameters could be important. For all experiments in this paper, we set $\alpha = \beta = 1$, giving types and ordering equal weights.

In the following explanation, C_i is a concrete use: a set of statements admitting a *happens before* relation (see Section IV-B). The selector function K determines if two concrete instances include two statements in the same order. The selector function T determines if two concrete instances use the same types in all relevant subexpressions of two statements.

$$\mathit{Dist}(C_1, C_2) = \sum_{\{i,j\} \in (C_1 \cup C_2)} \alpha K_{i,j}(C_1, C_2) + \beta T_{i,j}(C_1, C_2)$$

$$K_{i,j}(C_1, C_2) = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are } \textit{not} \text{ in the same order in } C_1 \text{ and } C_2 \\ 0 & \text{otherwise} \end{cases}$$

$$T_{i,j}(C_1, C_2) = \begin{cases} 1 & \text{if } i = j \text{ and } i \text{ and } j \text{ use different types in } C_1 \text{ and } C_2 \\ 0 & \text{otherwise} \end{cases}$$

Given this distance metric, we use the k-medoids algorithm [34] to cluster concrete uses (line 29). Because our distance metric must necessarily be computed between concrete objects, traditional k-means cannot be used: the k-means algorithm requires that distance can be computed between objects and also between arbitrary points in the metric space.

K-medoids takes a parameter: k, the number of clusters to find. k represents the inherent trade-off between precision and generality that exists with any documentation system. In our setting, a large value of k results in the algorithm returning multiple examples of the same basic usage pattern (e.g., the two uses of `BufferedReader` in Section II). If k is too small, however, it is possible that multiple usage patterns could be conflated. An overly large k can result is less representative examples, especially in terms of identifier names. All of the experiments in this paper use $k = 5$, since our initial study (see Section III-A) found no classes with more than four common patterns.

While previous approaches have applied clustering to code search (e.g., [12], [13]), to the best of our knowledge our approach is the first to leverage type information and statement ordering. We believe that these properties are essential to documenting many usage patterns (e.g., the best choice of data structure often depends on the usage pattern).

After clusters are discovered, the associated concrete uses are merged into *abstract uses* which are also represented as graphs. To merge a set of concrete uses, we first union all of the states (i.e., nodes) contained in each concrete use (line 31). The merging step is conceptually akin to the least-upper-bound lattice computation in constant-propagation dataflow analysis: $5 \sqcup 5 = 5$, but $x \sqcup y = \mathit{SOMETHING}$.

For example:

$$A = \left\{ \begin{array}{l} \mathit{stmt} : \texttt{print(x)} \\ \mathit{pred} : \texttt{x.isTrue()} \end{array} \right\} \qquad B = \left\{ \begin{array}{l} \mathit{stmt} : \texttt{return x} \\ \mathit{pred} : \texttt{x.isTrue()} \end{array} \right\}$$

$$A \sqcup B = \left\{ \begin{array}{l} \mathit{stmt} : \textit{Do something with } \texttt{x} \\ \mathit{pred} : \texttt{x.isTrue()} \end{array} \right\}$$

Transitions (i.e., edges) between statements in the abstract use are then constructed and weighted according to the number of concrete statements which include the transition (lines 32–36). Information about the types and names of parameters and return values is retained.

D. Emitting Documentation

The final stage of our algorithm transforms each abstract use into a representative, well-formed, and well-typed Java code fragment. Recall that abstract uses are graphs with edge weights that correspond to usage counts in the corpus (e.g., an edge with weight 15 connecting S and T indicates that there were 15 concrete uses which observed the ordering S *happens before* T).

An abstract use can contain both branches and cycles. Usage documentation, however, must be presented as linear text. Our next step is thus to find a representative ordering of the statements (i.e., a partial serialization) in each abstract use. We choose to approach this problem using a weighted topological sort. The goal is to find an ordering of state machine nodes (from start node to end node) which maximizes the sum of the weight of all outgoing edges minus the sum of the weight of all incoming edges. Such an ordering is optimal in the sense that any other ordering will be less representative of the concrete orderings in the corpus (i.e., more total statements must be swapped to align all of the concrete orderings with this this ordering).

More formally, we take as input set of statements S and function W, representing edge weights, which maps all pairs of statements in S to a value greater than or equal to 0. The task is to find a total ordering O, defined here as a function mapping a pair of unique statements (s_i, s_j) to 1 if and only if s_i comes before s_j in O and to -1 otherwise. The notion of representativeness can then be formalized by the Flow objective function below.

Input: Statement set $S : \{s_1, s_2, \ldots s_n\}$
Input: Edge weight mapping $W : (s_i, s_j) \rightarrow \{0 \ldots \infty\}$
Output: Ordering $O : (s_i, s_j) \rightarrow \{-1, 1\}$ maximizing:

$$\mathit{Flow} = \sum_{\{s_i, s_j\} \in S} O(s_i, s_j) \cdot W(s_i, s_j)$$

Our algorithm finds such an ordering by enumerating all possible orderings and computing Flow, returning the ordering for which Flow is largest (line 37). Note that the time complexity of this algorithm is factorial in the number of statements to be documented. However, this remains tractable for $n < 12$ and we have not encountered instances with $n > 8$. We implemented an unoptimized version of the algorithm and found that it was not a performance bottleneck in practice (see Section VI).

Once a statement ordering has been determined, the next step is to generate code (i.e., usage example documentation) from those ordered sets of statements (line 38). We desire code that is representative of the underlying concrete statements which were clustered into this abstract use. We first generate a basic block for each statement, guarded by its most common predicate (taken over the concrete uses forming this abstract use cluster). Statements are printed using standard Java syntax; names and types are assigned to all return values and parameters as needed. Adjacent blocks with the same predicate are merged (e.g., "`if (p) X; if (p) Y;`" becomes "`if (p) {X; Y;}`").

When choosing a name, we first check if any one name is used $X\%$ more often than all other names by human-written code in the corpus. In our experiments we set X to 100%, thus selecting a name if it occurs twice as often as any other name. If no such popular name exists, we use the declared formal parameter name for actual arguments and the first letter of the type name for other variables. For statements that are not invocations of the target class, we print "`//do something with Y`" where `Y` is the appropriate identifier. Types are chosen in the same manner, making use of declared return types for function return values. Declarations are inserted and marked with "`//initialized previously`" for all otherwise-undefined ("temporary") variables.

For each statement, we also count how often it was found inside a `try`, `catch`, or `finally` clause, and the type of the associated exception. If a statement is part of exception handling more often than not in the corpus, we then impose similar error handling in the generated example. In Java, well-formed exception handling involves a `try` block followed immediately by zero or more `catch` clauses and possibly a `finally` clause, with at least one `catch` or `finally` clause. All exception handling generated by our algorithm is of this form; we will not to generate a `catch` clause if there is no preceding `try`.

This construction ensures that no Java syntax rules are violated. Furthermore, we guarantee that all assignments are made to an identifier of the proper type and that all methods are invoked with the right number and types of arguments. It is important to note, however, that we cannot be sure that generated examples are free of other types of errors (e.g., run-time errors, unchecked exceptions, etc.).

The representativeness of our examples is well defined: if the order of any two statements were inverted (or if some statement were added or removed), then the resulting example would correspond to fewer real-word examples from the corpus. We are not aware of any previous approach to documenting APIs which offers such guarantees. Other correctness properties are, to some degree, orthogonal. For example, we could compile our examples and run them to check for run-time exceptions, or perhaps use model checking to verify temporal safety properties. While the topics are related, we are not proposing a static specification mining technique (where correctness is paramount); rather, we are proposing a common-usage documentation synthesis technique which could leverage work in specification mining. We view common behavior in the corpus as correct by definition for this documentation task.

Table I
CORPUS FOR EXAMPLE GENERATION IN THIS STUDY.

Name	Version	Domain	kLOC
FindBugs	1.2.1	Code Analysis	154
FreeCol	0.7.2	Game	91
hsqldb	1.8.0	Database	128
iText	2.0.8	PDF utility	145
jEdit	4.2	Word Processing	123
jFreeChart	1.0.6	Data Presenting	170
tvBrowser	2.5.3	TV Guide	138
Weka	3.5.6	Machine Learning	402
XMLUnit	1.1	Unit Testing	10
total			1361

V. INDICATIVE EXAMPLES

In this section we present a number of examples indicative of the strengths and limits of our approach. We constructed these examples using the program corpus enumerated in Table I, which includes nine popular open-source Java programs, as input to our prototype example generation tool. We issued example queries for each of the 47 SDK classes for which human-written example documentation was available (see Section III-A for details). Thirty-five of the classes were used at least once in our corpus and thus produced example documentation. After examining these examples in detail we performed a formal empirical evaluation and human study in Section VI and Section VII.

A. Naming

The names of types and variables are critical to producing readable, representative examples. The output of our tool can be sensitive to the types and variables used in the available corpus. Consider the following example for `java.util.Iterator`, which demonstrates how an `Iterator` can be used to enumerate arbitrary objects.

```
Iterator iter = SOMETHING.iterator();
while(iter.hasNext()) {
   Object o = iter.next();
   //do something with o
}
```

The same query, conducted using only the `FreeCol` benchmark, results in a very different example. Because use of the `EventIterator` class, which is a subclass of `java.util.Iterator`, is common in `FreeCol`, it is chosen as the most representative example.

```
EventIterator eventIterator =
      SOMETHING.eventIterator();
```

```
if(eventIterator.hasNext()) {
   Event e = eventIterator.nextEvent();
   //do something with e
}
```

We hypothesize that both behaviors are useful: the former constructs generic library API documentation while the later crafts specific documentation for internal APIs.

B. Patterns

Our algorithm finds and preserves common usage patterns: frequently occurring sequences of statements. However, there are many classes, such as the generic `Throwable`, for which there is no truly common pattern. The example our algorithms generates for `Throwable` shows that an instance might be queried for its *Class*, *Cause*, or *StackTrace*.

```
PrintWriter p; //initialized previously
Throwable e = SOMETHING.getThrown();
e.getClass();
e.getCause();
e.printStackTrace(p);
```

For classes such as this, more traditional API documentation which lists all the methods of a class and their functionality is clearly important. Furthermore, there are some patterns that cannot be captured by our algorithm. Interactions such as message passing patterns between multiple threads or client-server systems could not be discovered without additional analyses.

C. Exceptions

Ideally, examples should contain complete and correct exception handing. However, our tool learns actual exception handling from real code, which may not be fully correct. The example below of `java.io.ObjectInputStream` is one of a small number of classes for which our top-ranked example uses exception handling.

```
BufferedInputStream b;//initialized previously
ObjectInputStream stream =
    new ObjectInputStream(b);
try {
  Object o = stream.readObject();
  //Do something with o
} catch(IOException e) {
} finally {
  stream.close();
}
```

We view the correct use of exceptions and the correct handling of resources in the presence of exceptions as an orthogonal problem [23], [24], [25].

VI. Empirical Evaluation

In this section we begin the evaluation of our proposed algorithm and prototype implementation with two quantitative metrics: size and readability. Both of these features are considered very important by users of documentation (see Section III-B). Running our prototype tool on 1,361k lines of code to produce example documentation for 35 classes took 73 minutes (about 2 minutes per class). About 95% of this time is spent filtering the corpus and enumerating paths.

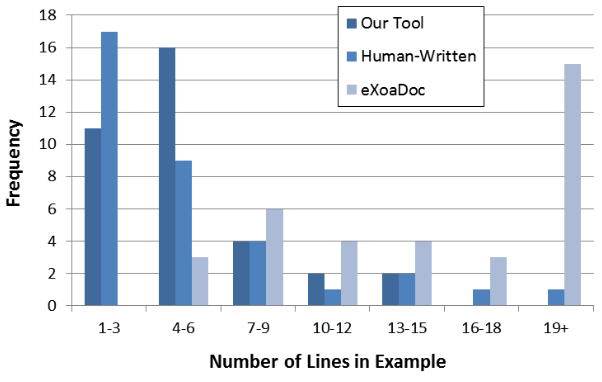

Figure 4. Size comparison (in lines of code) of examples generated by our tool using the corpus from Table I, written by humans, and mined by EXOADOC. The dataset is described in Section VI.

Throughout our evaluation we compare the output of our tool to both human-written examples from the Java SDK and also the EXOADOC tool of Kim *et al.* [13]. EXOADOC works by leveraging an existing code search engine to find examples of a class. Kim *et al.* then employ slicing to extract "semantically relevant" lines. These examples are then clustered and ranked based on *Representativeness*, *Conciseness* and *Correctness* properties. EXOADOC has been shown to increase productivity by as much as 67% in a small study.

Our dataset consists of examples from all 35 classes from standard Java APIs for which we have one example of each of the three types (see Section V). Because EXOADOCs are associated with methods rather than classes, we chose the top example for the most popular method (by static count of concrete uses in our benchmark set). For our tool, we chose the top (i.e., most representative) example for each class.

A. Size Evaluation

We compared the size (in lines of code) of each documentation type. The results are presented in Figure 4. Examples produced by our tool are slightly longer on average as compared to human written examples, but they follow approximately the same long-tail distribution. Much of the size difference can be attributed to our use of separate lines for "previously initialized" variables, which are often declared in-line by humans (see example in Section II).

Nonetheless, both human-written examples and our generated examples are shorter than EXOADOC examples, which are often longer than 19 lines. This difference is largely due to the less-relevant lines present in EXOADOC examples.

B. Readability Evaluation

Because readability was considered the most important characteristic of examples, we chose to evaluate the readability of the three documentation types directly. For this purpose we used an automated software readability metric [35]. This metric, which is based on and agrees with

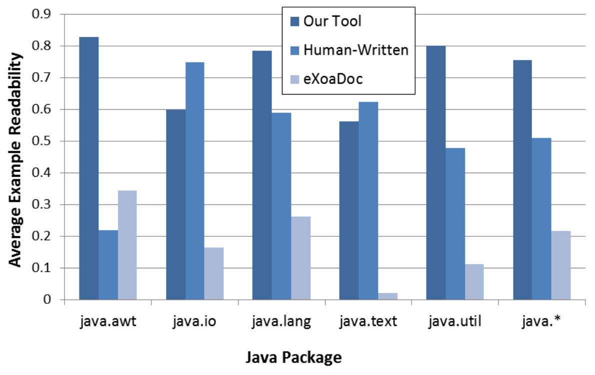

Figure 5. Readability comparison [35] of examples generated by our tool, written by humans, and mined by EXOADOC. The scale ranges from 1.0 (most readable) to 0.0 (least). The dataset, which we breakdown into 5 Java packages, is described in Section VI.

a large set of human judgments, reports Java code snippet readability on a scale of 0 (least readable) to 1 (most).

Figure 5 presents the results of that analysis. Overall, examples generated by our tool are about 25% more readable than human-written examples and over 50% more readable than examples mined with EXOADOC (t-test significance level < 0.002 in both cases). This difference is consistent across the major Java package areas to which the documented classes belong. Additionally, generated documentation is more consistently readable: the standard deviation in the readability score of our tool output is about 40% lower than that of the other two documentation types.

VII. HUMAN STUDY

In this section we present a human study of API example quality. The goal of this study is to quantify the desirability of the output of our tool in comparison to both human-written examples (from the Java SDK) and the state-of-the-art tool EXOADOC of Kim *et al.* [13]. The study involved 154 participants evaluating 35 pairs of API examples.

A. Experimental Setup

For each of the 35 classes from the dataset described in Section VI, the participant is shown the name of the target class and is randomly shown two of the three documentation types (i.e., ours, EXOADOC, and human-written). The participant is then required to make a preference evaluation by selecting one option from a five-element Likert scale: "Strong Preference" for A, "Some Preference" for A, "Neutral", "Some Preference" for B, "Strong Preference" for B. If desired, the participant may also choose to "Skip" the pair.

The study was advertised to students at The University of Virginia enrolled in a class entitled *Software Development Methods*, which places a heavy focus on learning the Java language and its standard set of APIs. For comparison, 16 computer science graduate students also participated. 179 students participated in total, however, to help preserve data integrity, we removed from consideration the results from 25 undergraduate students who completed the study in less than five minutes. The average time to complete the study for the remaining 154 participants was 13 minutes.

Participation was voluntary and anonymous. The participants were instructed to "Pretend that [they] are a programmer or developer who needs to use, or understand how to use, the class." No additional guidance was given; participants formed their own opinions about each example.

B. Quantitative Results

In total, 154 participants compared 35 example pairs each, producing 5,390 distinct judgments. The aggregate results are presented in Figure 6. Overall, the output of our tool was judged at least as good as human written examples over 60% of the time and strictly better than EXOADOC in about 75% of cases. For 82% of examples, on average either humans preferred our generated documentation to human-written examples or had no preference. For 94% of examples, the output of our tool was preferred to EXOADOCS.

Raw score distributions were very similar between graduate and undergraduate students. However, taken example by example, grad students had a 20% reduced preference for tool-generated examples when compared to human-written examples. Nonetheless, both grads and undergrads judged our generated documentation to be at least as good as gold-standard human-written for over half of the examples. Compared to EXOADOC, our tool was preferred in almost all cases by both groups.

Finally, we asked whether "a program that automatically generates examples like these would be useful?", and 81% agreed that it would be either "Useful" or "Very Useful."

VIII. THREATS TO VALIDITY

Although our experiments suggest that our algorithm produces examples that are preferred to state-of-the-art code search techniques and as good as human-written documentation over 80% of the time, our results may not generalize.

First, the benchmarks we selected may not be indicative. We mitigated this threat by performing our human evaluation on documentation for standard library classes — classes used in almost every program and by almost every programmer. It is possible, however, that results for these classes may not generalize to all APIs. However, the documented classes did feature a wide range of usage patterns. In addition, the output of our tool was shown to be consistently good over all major packages in the Java SDK.

Our use of students for evaluation may not generalize across all populations and to professional developers in particular. This threat is somewhat mitigated by the observation that our graduate student population agreed quite closely with our undergraduate one despite the fact that the graduates had significantly varying backgrounds and typically had more years of programming experience.

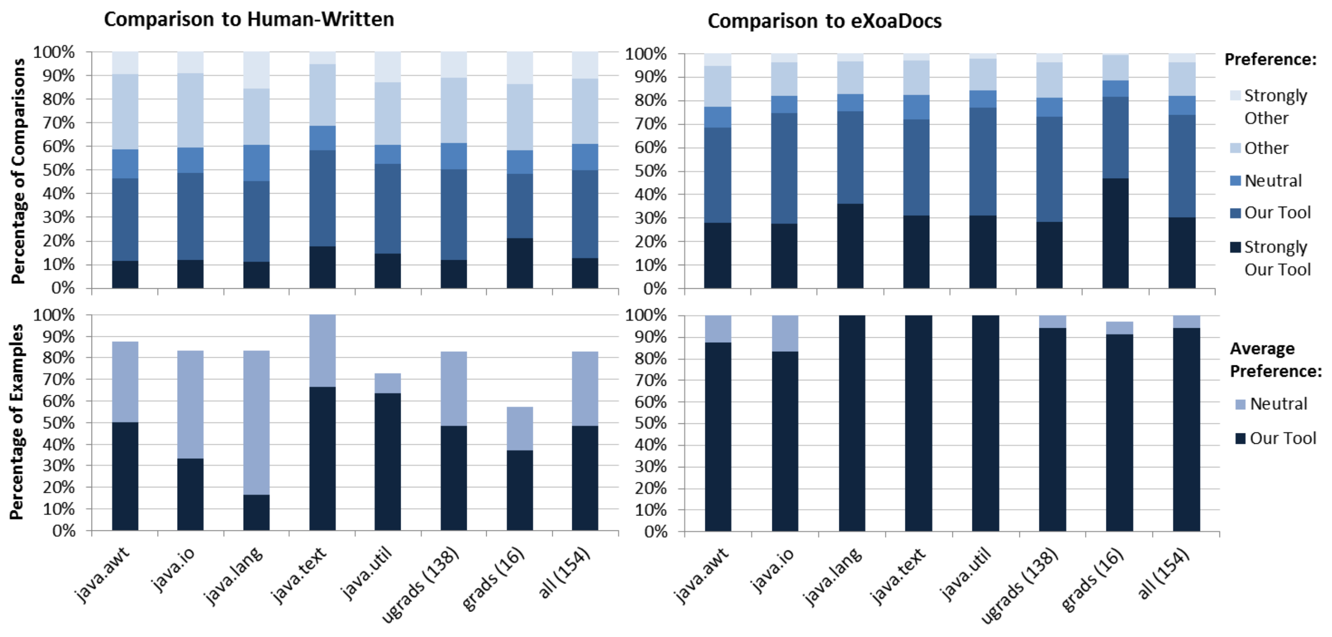

Figure 6. Aggregated results from our human study. The left charts show responses comparing our tool to human-written examples and the charts on the right compare our tool to the EXOADOC tool of Kim *et al.* [13]. The charts on the top categorize all comparisons (e.g., in 60% of comparisons, our tool output was judged at least as good as human-written examples). The charts on the bottom reflect the consensus opinion for each example (e.g., 82% of examples produced by our tool were judged to be at least as good as human-written on average). Examples are grouped by package (e.g., `java.X`) and by category of participants (138 ugrads, 16 grads).

IX. Related Work

The most closely related work is that of Kim *et al.* [13] and Zhong *et al.* [12]. We compare to Kim *et al.* directly and have discussed Zhong *et al.* previously. We now briefly describe other works related to mining API usage patterns and their documentation.

Our technique is inspired in part by the work of Whaley *et al.* [32] who, in 2002, used multiple finite state machine submodels to model the interface of a class. They used dynamic instrumentation techniques to extract such models from execution runs. Whaley *et al.* note that these models can potentially serve as documentation, however they do not evaluate the efficacy of such an approach.

Wasylkowski *et al.* [36] adapted static specification mining techniques to learn common, but not required, method-call sequences. Similar to our technique, they use state machines to represent common usage patterns for an object. Nguyen *et al.* [37] employ a similar approach to mine usage patterns, but across multiple objects. Both approaches focus on defect and anomaly detection. By contrast, our work focuses on generating and evaluating human-readable documentation, assuming that average behavior in the input corpus is correct.

Yessenov *et al.* [38] present a tool called MATCHMAKER which synthesizes code suitable for use as an example. A MATCHMAKER query consists of names of two APIs, and produces code enabling interaction between them. Unlike our approach which abstracts from static usage examples, MATCHMAKER relies on a database of dynamic program traces to reason about the evolution of the heap that connects the two APIs.

Dekel *et al.* [39] decorate method invocations with rules or caveats of which client authors must be aware. The technique increases awareness of important documentation written by humans. Holmes *et al.* [17] use structural context to recommend source code examples. Their approach finds relevant code in an example repository by heuristically matching the structure of the code under development.

X. Conclusion

API examples are known to be a key learning resource, however they are expensive to create and are often unavailable or out-of-date. Current tools for mining examples produce output that is complex and difficult to understand, compromising usefulness. These observations led us to propose an algorithm for *generating* API usage examples. Our technique is efficient and fully automated; to the best of our knowledge, it is the first approach that synthesizes *human-readable* documentation for APIs. Our output is correct-by-construction in several important respects (e.g., syntactically valid and type-safe) and adheres to a well-defined notion of representativeness: if the order of any two statement in one of our examples were inverted, the result would correspond to fewer real-world uses. In a human study involving over 150 participants, we have shown it to produce output judged at least as good as gold-standard human-written documentation 82% of the time and strictly better than a state-of-the-art code search tool 94% of the time.

References

[1] P. Hallam, "What do programmers really do anyway?" in *Microsoft Developer Network — C# Compiler*, Jan 2006.

[2] S. L. Pfleeger, *Software Engineering: Theory and Practice*. NJ, USA: Prentice Hall, 2001.

[3] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *International Conference on Design of Communication*, 2005, pp. 68–75.

[4] D. G. Novick and K. Ward, "What users say they want in documentation," in *Conference on Design of Communication*, 2006, pp. 84–91.

[5] R. Holmes, R. Cottrell, R. Walker, and J. Denzinger, "The end-to-end use of source code examples: An exploratory study," in *International Conference on Software Maintenance*, 2009, pp. 555–558.

[6] R. Jain, "API-writing and API-documentation," in *http://api-writing.blogspot.com/*, Apr. 2008.

[7] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, "Building more usable apis," *IEEE Softw.*, vol. 15, no. 3, pp. 78–86, 1998.

[8] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon, "What programmers really want: results of a needs assessment for sdk documentation," in *International Conference on Computer documentation*, 2002, pp. 133–141.

[9] F. Shull, F. Lanubile, and V. R. Basili, "Investigating reading techniques for object-oriented framework learning," *IEEE Trans. Softw. Eng.*, vol. 26, no. 11, pp. 1101–1118, 2000.

[10] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: improving api documentation using usage information," in *Extended Abstracts on Human Factors in Computing Systems*, 2009, pp. 4429–4434.

[11] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE Softw.*, vol. 26, no. 6, pp. 27–34, 2009.

[12] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *ECOOP*, 2009, pp. 318–343.

[13] J. Kim, S. Lee, S. Hwang, and S. Kim, "Towards an intelligent code search engine," in *AAAI Conference on Artificial Intelligence*, 2010.

[14] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Principles of programming languages*, 2002, pp. 4–16.

[15] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *Programming Languages Design and Implementation*, 2005, pp. 48–61.

[16] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Automated Software Engineering*, 2007, pp. 204–213.

[17] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *International Conference on Software Engineering*, 2005, pp. 117–125.

[18] javadoctool@sun.com, "How to write doc comments for the Javadoc tool," in *http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html*, 2010.

[19] Oracle, "Java SE 6 documentation," in *http://download.oracle.com/javase/6/docs/*, 2010.

[20] R. Chatterjee, B. G. Ryder, and W. Landi, "Complexity of points-to analysis of java in the presence of exceptions." *IEEE Trans. Software Eng.*, vol. 27, no. 6, pp. 481–512, 2001.

[21] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for the analysis of java programs," in *Workshop on Program Analysis for Software Tools and Engineering*, 1999, pp. 21–31.

[22] S. Sinha, A. Orso, and M. J. Harrold, "Automated support for development, maintenance, and testing in the presence of implicit control flow," in *International Conference on Software Engineering*, 2004, pp. 336–345.

[23] W. Weimer and G. C. Necula, "Finding and preventing runtime error handling mistakes," in *Conference on Object-oriented programming, systems, languages, and applications*, 2004, pp. 419–431.

[24] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 191–221, 2003.

[25] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of Java exceptions using JESP," in *Int. Conf. on Compiler Construction*, 2000, pp. 67–81.

[26] P. A. Relf, "Tool assisted identifier naming for improved software readability: an empirical study," *Empirical Software Engineering*, November 2005.

[27] M. Das, S. Lerner, and M. Seigle, "ESP: path-sensitive program verification in polynomial time," *SIGPLAN Notices*, vol. 37, no. 5, pp. 57–68, 2002.

[28] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante, "Path analysis and renaming for predicated instruction scheduling," *International Journal of Parallel Programming*, vol. 28, no. 6, pp. 563–588, 2000.

[29] T. Robschink and G. Snelting, "Efficient path conditions in dependence graphs," in *International Conference on Software Engineering*, 2002, pp. 478–488.

[30] R. Alur, P. Cerny, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for Java classes," in *Principles of Programming Languages*, 2005.

[31] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2005, pp. 461–476.

[32] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *International Symposium of Software Testing and Analysis*, 2002.

[33] S. E. Stemler, "A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability," *Practical Assessment, Research and Evaluation*, 2004.

[34] L. Kaufman and P. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Interscience, 1990.

[35] R. P. L. Buse and W. R. Weimer, "A metric for software readability," in *International Symposium on Software Testing and Analysis*, 2008, pp. 121–130.

[36] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Foundations of Software Engineering*, 2007, pp. 35–44.

[37] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Foundations of Software Engineering*, 2009, pp. 383–392.

[38] K. Yessenov, Z. Xu, and A. Solar-Lezama, "Data-driven synthesis for object-oriented frameworks," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 65–82. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048075

[39] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *International Conference on Software Engineering*, 2009, pp. 320–330.