Is This Class Thread-Safe? Inferring Documentation using Graph-Based Learning

Andrew Habib
andrew.a.habib@gmail.com
Department of Computer Science
TU Darmstadt
Germany

Michael Pradel
michael@binaervarianz.de
Department of Computer Science
TU Darmstadt
Germany

ABSTRACT

Thread-safe classes are pervasive in concurrent, object-oriented software. However, many classes lack documentation regarding their safety guarantees under multi-threaded usage. This lack of documentation forces developers who use a class in a concurrent program to either carefully inspect the implementation of the class, to conservatively synchronize all accesses to it, or to optimistically assume that the class is thread-safe. To overcome the lack of documentation, we present TSFinder, an approach to automatically classify classes as supposedly thread-safe or thread-unsafe. The key idea is to combine a lightweight static analysis that extracts a graph representation from classes with a graph-based classifier. After training the classifier with classes known to be thread-safe and thread-unsafe, it achieves an accuracy of 94.5% on previously unseen classes, enabling the approach to infer thread safety documentation with high confidence. The classifier takes about 3 seconds per class, i.e., it is efficient enough to infer documentation for many classes.

CCS CONCEPTS

• **Software and its engineering** → **Synchronization**; **Reusability**; **Documentation**; *Automated static analysis*; *Object oriented development*; Software maintenance tools;

KEYWORDS

inferring documentation, thread-safe class, graph-based learning

ACM Reference Format:
Andrew Habib and Michael Pradel. 2018. Is This Class Thread-Safe? Inferring Documentation using Graph-Based Learning. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238212

1 INTRODUCTION

Thread-safe classes are pervasive. They are the corner stone of concurrent, object-oriented programs. A thread-safe class encapsulates all necessary synchronization required to behave correctly when its instances are accessed by multiple client threads concurrently, without additional synchronization from the calling side. Developers of multi-threaded object-oriented programs often rely on thread-safe classes to cast away the burden of ensuring the thread safety of their applications.

Unfortunately, it is not always clear to a developer who uses a class whether the class is thread-safe or not. The reason is that many classes do not provide any or only partial information about their thread safety. Instead, it is common to find questions on web forums, such as Stack Overflow, about the thread safety of a specific class. For example, one developer asked about the thread safety of the widely used `javax.xml.parsers.DocumentBuilder` class.[1] Another developer questioned the thread safety of the crucial JDK class `java.util.Random`.[2] Developers often complain about the lack of thread-safety documentation. For instance, the developer who reported that earlier versions of JDK format classes are not thread-safe notes that: "Not being thread-safe is a significant limitation on a class, with potentially dire results, and not documenting the classes as such is dangerous."[3] Eventually, the accepted fix was to explicitly state in the documentation that JDK format classes are not thread-safe. Another developer comments on the thread-safety of `java.beans.PropertyChangeSupport` and `java.beans.VetoableChangeSupport` and writes: "[...] However, the documentation does not indicate either their thread-safety or lack thereof. In keeping with the current documentation standards, this point should be indicated in the class documentation."[4]

The lack of adequate documentation about the thread safety of classes has several negative consequences. First, a developer may solve the problem by manually analyzing the classes she wants to reuse. However, this approach spoils some of the benefits of reusing an existing class because it forces the developer to inspect and understand the class implementation, breaking the encapsulation provided by the class API. Second, a developer may conservatively assume that a class is not thread-safe and carefully synchronize all concurrent accesses to the class to avoid concurrency bugs, such as data races, atomicity violations, and deadlocks. However, if the class is already thread-safe, this additional synchronization imposes additional runtime overhead and may unnecessarily limit the level of parallelism achieved by the program. Finally, a developer may optimistically assume a class to be thread-safe. However, if the class turns out to not provide this guarantee, the program may suffer from concurrency bugs, e.g., [1–3], which often become apparent

[1] https://www.stackoverflow.com/questions/56737
[2] https://www.stackoverflow.com/questions/5819638
[3] https://bugs.java.com/view_bug.do?bug_id=4264153
[4] https://bugs.java.com/view_bug.do?bug_id=5026703

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '18, September 3–7, 2018, Montpellier, France
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238212

only under specific interleavings and therefore may easily remain unnoticed during testing. In all three scenarios, the developer takes a poorly guided decision that relies on her limited understanding of an implementation or on luck.

This paper addresses the problem of missing thread safety documentation by automatically classifying a given class as thread-safe or thread-unsafe. Our approach, called TSFinder, is a statistical, graph-based learning technique that learns from a relatively small set of classes known to be thread-safe or thread-unsafe the distinguishing properties of these two kinds of classes. The approach is enabled by two contributions. First, TSFinder uses a lightweight static analysis of the source code of the class to extract information and represent this information in a graph. Second, we use graph-based classification techniques – graph kernels [65] combined with support vector machine (SVM) [56] to learn a classifier for previously unseen classes. TSFinder helps developers assess the thread safety of an otherwise undocumented class, enabling a developer to take an informed decision on whether and how to use the class.

Our work is complementary to techniques for finding concurrency bugs, which has been extensively studied in the past [7, 13, 18, 19, 35, 36, 42, 55, 68, 71, 78], in particular in the context of thread-safe classes [14, 41, 44, 51–53, 61]. These approaches consider supposedly thread-safe classes and try to find corner cases in their implementation that a developer has missed. Instead, TSFinder addresses classes for which it is unknown whether the class is even supposed to be thread-safe and tries to answer that question in an automatic way. Applying existing bug detection techniques to answer this question would likely result in missing thread-unsafe classes (by testing-based approaches) or missing thread-safe classes (by sound static analyses). Our work also relates to existing work on inferring [5, 26] and improving [27, 37, 63, 81] documentation. We extend this stream of work to concurrency-related documentation, which has not yet been studied.

Our evaluation consists of two parts. First, we validate our hypothesis that existing classes lack thread safety documentation by systematically searching all 179,239 classes in the Qualitas corpus [60]. We find that the vast majority of classes fails to document whether it is thread-safe or not. Second, we evaluate our classifier with 230 training classes that were manually labeled as thread-safe or thread-unsafe. We find that 94.5% of TSFinder's classification decisions are correct. In particular, the precision and recall of identifying thread-safe classes are 94.9% and 94.0%, respectively. On average, adding documentation to a new class takes about 3 seconds. These results show that the approach is accurate enough to significantly improve over guessing whether a class is thread-safe and efficient enough to scale to large sets of classes.

In summary, this paper makes the following contributions:

- A systematic study of thread-safety documentation in real-world Java classes showing the lack of such documentation.
- The first automated classifier to distinguish supposedly thread-safe and thread-unsafe classes, an understudied problem that addresses the lack of thread safety documentation.
- A novel combination of static analysis and graph-based classification that accurately and efficiently predicts the thread safety of a class.

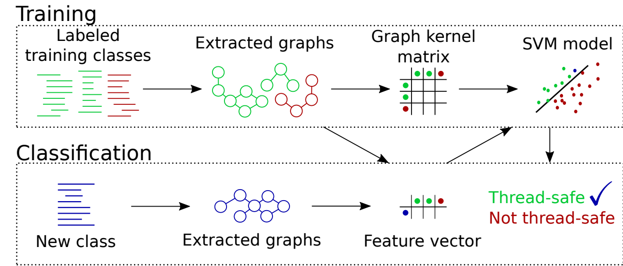


Figure 1: Overview of TSFinder: Inferring thread safety using static analysis and graph kernels.

In Section 2, we give an overview of TSFinder. Sections 3 and 4 fill in the details. Sections 5 and 6 summarize the implementation and evaluation. Sections 7 and 8 discuss the limitations of TSFinder and and related work. Finally, in Section 9 we conclude the paper and discuss future work.

2 CHALLENGES AND OVERVIEW

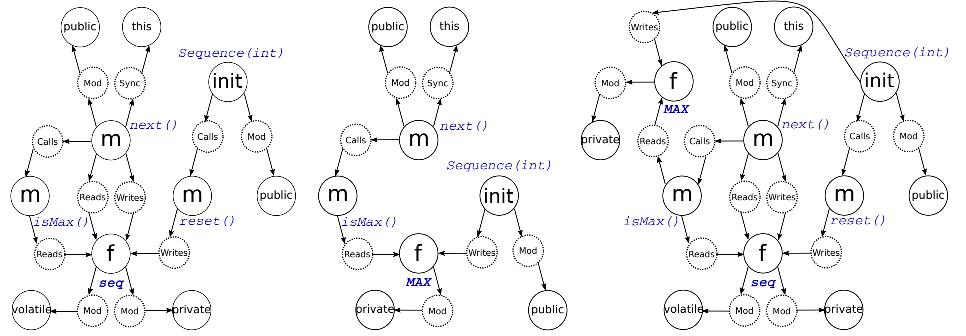
The goal of this work is to automatically document classes as supposedly thread-safe or thread-unsafe. The approach should be efficient enough to scale to hundreds of classes, e.g., all classes in a 3rd-party library, and accurate enough to provide reliable documentation. Achieving this goal is challenging for several reasons. First, there are different approaches for ensuring that a class is thread-safe, e.g., making the class immutable, using language-level synchronization primitives, building on other thread-safe classes, using lock-free data structures, and combinations of these approaches. Because of this diversity, a simple check, e.g., for whether a class has synchronized methods, is insufficient to determine thread safety. Second, the thread safety of a class may depend on other classes. In particular, inheriting from a thread-unsafe class may compromise the thread safety of the child class. Third, extensive reasoning about concurrent behavior, e.g., to reason about different interleavings [66], can easily require large amounts of computational resources, which conflicts with our scalability goal.

Figure 1 provides an overview of our approach to infer thread safety documentation. The approach consists of a training phase, where it learns from a set of classes known to be thread-safe and thread-unsafe, and a prediction phase, where it infers thread safety documentation for a previously unseen class. Both phases combine a lightweight static analysis that extracts graph representations of classes with a graph-based classifier. The graph-based classification converts graphs into vectors by computing the similarity between graphs of a to-be-classified class and graphs in the trained model. These vectors are then classified using a model based on a support vector machine (SVM). The following illustrates the main steps of TSFinder using the Java class in Figure 2a.

Extracting field-focused graphs. To apply machine learning to the thread safety classification problem, we need to represent classes in a suitable form. Our approach exploits the structured nature of programs by representing a class as a set of graphs. Since multi-threading is mainly about sharing and allowing multiple concurrent

```
public class Sequence {
  private volatile int seq;
  private int MAX;
  public Sequence(int m) {
    MAX = m;
    reset();
  }
  public synchronized
      int next() {
    if(!isMax())
      return seq++;
    return -1;
  }
  boolean isMax() {
    return seq > MAX;
  }
  void reset() {
    seq = 0;
  }
}
```

(a) Java class.



(b) Extracted graphs. The graphs from left to right correspond to fields seq, `MAX`, and the pair (seq, `MAX`), respectively. The identifier names, in italic and blue font, are not used for classification, but shown only for illustration.

$$\begin{matrix} g_1 & g_2 & \cdots & g_{4860} \\ [0.350 & 0.436 & \ldots & 0.573] \end{matrix} \quad \begin{matrix} g_1 & g_2 & \cdots & g_{4860} \\ [0.355 & 0.536 & \ldots & 0.584] \end{matrix} \quad \begin{matrix} g_1 & g_2 & \cdots & g_{4860} \\ [0.392 & 0.588 & \ldots & 0.567] \end{matrix}$$

(c) Vectors of the three graphs in Figure 2b. The trained model has 4,860 graphs in it.

$$\begin{matrix} min(g_1) & max(g_1) & avg(g_1) & min(g_2) & max(g_2) & avg(g_2) & \cdots & avg(g_{4860}) \\ [0.350 & 0.392 & 0.366 & 0.436 & 0.588 & 0.520 & \ldots & 0.575] \end{matrix}$$

(d) Class vector of the entire class.

Figure 2: A Java class and graphs extracted from it by our analysis. TSFinder predicts this class to be *thread-safe*.

accesses to resources, the graphs represent shared resources and how these resources are accessed.

For the example class, Figure 2b shows the graphs extracted by TSFinder. Each graph focuses on a single field or a combination of fields of the class. The graphs represent read and write accesses to the fields, call relationships between methods, and the use of synchronization primitives, such as the `synchronized` keyword. For example, the first graph in Figure 2b which focuses on the `seq` field shows that the field is read by the `isMax` method, written by the `reset` method, and both read and written by the `next` method. Furthermore, the graph represents the call relationship between `next` and `isMax`.

Computing graph kernels. After extracting a set of graphs for each class under analysis, TSFinder checks for each graph whether it is similar to graphs that come from thread-safe or from thread-unsafe classes. To this end, we use the graph kernels [65], i.e., mathematical functions that compute the pairwise similarity of graphs. TSFinder computes the similarity of each graph of a class and the graphs of classes known to be thread-safe or thread-unsafe. The similarity values yield a vector of numbers, called the *graph vector* or *embedding*. For the running example, the approach computes three graph vectors, one for each graph, as illustrated in Figure 2c.

Learning a classification model. To train a classifier that can distinguish thread-safe classes from thread-unsafe classes, TSFinder trains a model using a corpus of classes with known thread safety. The approach combines all graph vectors of a class into a single vector, called *class vector*, that represents the entire class (Figure 2d) along with a label denoting whether the class is thread-safe or not. Finally, the labeled class vectors are used to train a classification model that distinguishes between the two kinds of classes.

Classifying a new class. Given a new class, our approach extracts graphs and computes a class vector as in the previous step. Based on the trained model, TSFinder then classifies the class by querying the model with this vector. For Figure 2, TSFinder infers that the class is *thread-safe* and adds this information to the class documentation.

3 EXTRACTING FIELD-FOCUSED GRAPHS

The first step of our approach is to extract graphs from classes via a lightweight static analysis. This section explains the properties extracted by the static analysis (§ 3.1) and how we summarize these properties into graphs (§ 3.2).

3.1 Static Analysis

TSFinder performs a lightweight static analysis that extracts various properties of a class under analysis. We focus on two kinds of properties: *unary properties*, which describe program elements of the class, and *binary properties*, which describe relationships between program elements and properties of program elements. We choose properties relevant for concurrency, e.g., memory locations, accesses to memory locations, and memory visibility guarantees of these accesses.

3.1.1 Unary Properties.

The static analysis extracts the following unary properties from each class:

Definition 3.1 (Unary properties). Let C be the class under analysis. Let C_f be the set of fields, C_m be the set of methods, and C_{const} be the set of class constructors and static constructors defined by C. The set of unary properties of C is:

$$C_{unary} = C_f \cup C_m \cup C_{const}$$

For example, our approach extracts from the class in Figure 2a the following set of unary properties:

$$C_f = \{seq, MAX\}, C_m = \{next(), isMax(), reset()\}$$

and

$$C_{const} = \{Sequence(int)\}$$

3.1.2 Binary Properties.

To capture relationships between program elements and properties of program elements, the analysis extracts several binary properties:

Definition 3.2 (Binary properties). Let C be the class under inspection, and C_{const}, C_m, C_f as defined above. We define the following binary relations $Rels$:

- $Calls : \{C_{const} \cup C_m\} \times \{C_{const} \cup C_m\}$
- $Reads : \{C_{const} \cup C_m\} \times \{C_f\}$
- $Writes : \{C_{const} \cup C_m\} \times \{C_f\}$
- $Sync : \{C_m\} \times \{this, lock\}$
- $Mod : \{C_{const} \cup C_m \cup C_f\} \times \{public, protected, private, static, volatile, final\}$

The set of binary properties of C is:

$$C_{binary} = Calls \cup Reads \cup Writes \cup Sync \cup Mod$$

The binary properties capture a rich set of relations relevant to our thread safety prediction task, e.g., whether a method is public, what fields a method reads and writes, and whether a method is synchronized. The set $\{this, lock\}$ represents objects that the class uses as locks, where $this$ represents a self-reference to the current instance and $lock$ represents any other object.

For our running example in Figure 2a, the binary properties include that the public class constructor `Sequence(int)` writes to the field `MAX`, that the method `next()` reads and writes the field `seq`, and that the method `next()`is `synchronized` on `this`. Note that the absence of properties also conveys information. For example, the absence of a binary relation $(MAX, volatile) \in Mod$ indicates that the `MAX` field is non-volatile.

3.1.3 Flattening the Class Hierarchy.

The thread safety of a class not only depends on its own implementation, but also on the implementation of its superclasses. E.g., a class may inherit a method that does not synchronize data accesses and therefore become thread-unsafe, even though the subclass alone would be thread-safe [45]. Our static analysis addresses this challenge by flattening the class hierarchy. Specifically, the analysis recursively merges the unary and binary properties of each class with those of its superclass until reaching the root of the class hierarchy. The merging follows the inheritance rules of the Java language. For example, the properties related to a superclass method that is not overridden by the subclass are merged into the properties of the subclass.

3.2 Field-focused Graphs

Given the properties extracted by the static analysis, TSFinder summarizes this information into a set of graphs for each class. Traditionally, programs have been represented by a variety of graphs suited for different purposes. For example, abstract syntax trees, control-flow graphs, and program dependency graphs have been used to analyze the syntax, control flow, and data flow of programs. The following presents two kinds of graphs designed specifically to reason about concurrency-related properties of classes. The basic idea is to summarize in each graph how clients of the class may access a field or a combination of fields of the class. We call these graph representations *field-focused graphs*.

Before presenting field-focused graphs, we define a single graph per class, which conflates all properties known about this class:

Definition 3.3 (Class graph). Given a class C, let C_{unary} and C_{binary} be the unary and binary properties of C, respectively. The class graph of C is a directed multi-graph $g_C = (V_C, E_C)$, where $V_C = V_{Rels} \cup C_{unary} \cup \{this, lock, public, protected, private, static, volatile, final\}$ are vertices that represent program elements and properties of them, and $V_{Rels} = \{Calls, Reads, Writes, Sync, Mod\}$ are special nodes that represent the different relations in C_{binary}. Each special node is labeled with the name of the relation, i.e., with $Calls$, $Reads$, $Writes$, $Sync$, or Mod and is connected to its binary operands by the set E_C of directed unlabeled edges.

One possible approach would be to predict the thread safety of a class based on its class graph. However, most class graphs are dissimilar from most other class graphs, independently of whether the classes are thread-safe, because classes and therefore also their class graphs are very diverse. An important insight of our work is that this problem can be addressed by deriving smaller graphs from the class graph, so that each small graph captures a coherent subset of concurrency-related properties. The intuition is that these smaller graphs capture recurring implementation patterns of thread-safe and thread-unsafe classes, enabling TSFinder to learn to distinguish them.

TSFinder derives smaller graphs from the class graph by focusing on a single field or a combination of fields:

Definition 3.4 (Field-focused graph). Given a non-empty subset $F \subseteq C_f$ of the fields of a class C and a class graph g_C where $g_C = (V_C, E_C)$, the field-focused graph $g_F = (V_F, E_F)$ contains all vertices reachable from F, i.e., $V_F = \{v \mid \exists v_f \in F \text{ s.tt. } reachable_{g_C}(v_f, v) \text{ and } reachable_{g_C}(v, v_f)\}$, and contains all edges connecting these vertices.

For a directed graph $g = (V, E)$ where u and $v \in V$,

$$reachable_g(u, v) \iff \text{there exists a directed edge from } u \text{ to } v.$$

If the set F contains a single field, then the field-focused graph captures all program elements related to this field, as well as the relations between them. Such a single-field graph summarizes how clients of the class may access the field and to what extent these accesses are protected by synchronization.

For the example in Figure 2a, TSFinder extracts two graphs that focus on single fields, shown as the first two graphs in Figure 2b. They focus on the fields `seq` and `MAX`, respectively.

Some characteristics of thread-safe classes cannot be captured by single-field graphs. For example, a thread-safe class may update two semantically related fields together and use a single lock or a synchronized method to protect the access to these fields. TSFinder captures such behavior by also considering sets F of multiple fields, which yields multi-field graphs. Specifically, the approach considers all pairs of fields for which there exists at least one method that reads or writes from both fields. To bound the overall number of graphs extracted per class, we focus on field-focused graphs with $|F| \leq 2$, i.e., single fields or pairs of fields.

As an example of a multi-field graph, consider the third graph in Figure 2b. Because the class method `isMax()` reads both fields, the approach extracts a graph that captures both fields together.

Intuitively, the reason why field-focused graphs are effective at characterizing the thread safety of a class is that they capture various patterns for making a class thread-safe. Whether a class is thread-safe depends on how the class accesses its internal state, i.e., its fields, and in what ways these accesses are protected by synchronization. Field-focused graphs capture the various ways to implement thread safety, e.g., using synchronized methods, volatile fields, or by making a class immutable. By capturing these implementation patterns, the graphs enable TSFinder to determine whether a class is thread-safe.

Graph canonicalization. The final step of extracting field-focused graphs from classes is to canonicalize the graphs. The motivation is that, to learn recurring patterns in implementations of thread-safe and thread-unsafe classes, the extracted graphs need to be comparable across different classes. In particular, they should not contain identifier names, such as method and field names, as these vary across different classes and projects. Therefore, our approach renames each node that represents a method to `m`, while two special node names `init` and `clinit` are reserved for class constructors and static constructors, respectively. Similarly, all fields are renamed to `f`.

4 CLASSIFYING CLASSES

Classifying graphs is a classical problem in several domains such as bio- and chemo-informatics [10, 47, 59], image analysis [25], and web and social network analysis [67]. Traditional approaches to this problem [65] use a so-called kernel method [56], a function to compute the similarity between two graphs. The pairwise similarities between graphs are then used as vector embeddings to represent the graphs for classification.

We adopt a variant of this approach to our problem of classifying thread-safe classes. TSFinder first builds several graphs per class (§ 3.2). It then uses the kernel method through a graph kernel function to generate vectors (embeddings) for graphs (§ 4.2.1). Instead of training a machine learning model on several individual graphs from each class, we combine embeddings of graphs extracted from the same class into one single embedding per class for the machine learning model to learn (§ 4.2.2). This step allows TSFinder to classify thread-safe classes using all generated graphs from a class.

Based on the field-focused graphs extracted for each class, TSFinder learns how to classify classes into supposedly thread-safe and thread-unsafe classes. To this end, the approach combines a graph kernel, which computes the similarity of two graphs, with a SVM, which classifies each class based on the similarity of its graphs to other graphs from classes known to be thread-safe or not.

The basic idea is to perform three steps:

(1) Given a class, compare its graphs to graphs of classes known to be thread-safe or thread-unsafe. For each pair of graphs, compute a similarity score and summarize all scores into a vector per graph.
(2) Combine all graph vectors of a class into one single class vector that summarizes the similarity of graphs extracted from the class to graphs in the trained model.
(3) Classify a class by querying a vector-based binary classifier using the resulting class vector. The classifier has been trained with the class vectors of the classes with known thread safety.

The remainder of this section presents these steps in detail.

4.1 Background: Graph Kernels

Checking whether two graphs are isomorphic is a computationally hard problem for which no polynomial-time algorithm is known. In contrast, graph kernels offer an efficient alternative that compares graph substructures in polynomial time. In essence, a graph kernel is a function that takes two graphs and yields a real-valued similarity score. Given a list of graphs $g_1, .., g_n$ and a kernel k, one can compute a matrix $K = (k(g_i, g_j))_{i,j}, 1 \leq i, j \leq n$, that contains all pairwise similarity scores of the graphs. This matrix, called the *kernel matrix*, is symmetric and positive-definite.

In this work, we build upon a fast, scalable, state of the art kernel, the *Weisfeiler-Lehman (WL) graph kernel* [58]. It is based on the Weisfeiler-Lehman graph isomorphism test [74], which augments each labeled node by the sorted set of its direct neighbors and compresses this augmented label into a new label. This step is repeated until the sets of node labels of the two graphs are different or until reaching a maximum number of iterations h. Given a graph g, we refer to the sequence of graphs obtained by this augmentation and compression step as $g_0, g_1, .., g_h$, where $g_0 = g$ and g_h is the maximally augmented and compressed graph. We call this sequence of graphs the *WL sequence* of g.

Given two graphs and their WL sequences, we compute the graph kernel as follows:

Definition 4.1 (Weisfeiler-Lehman kernel). The graph kernel of g and g' is

$$k(g, g') = k_{sub}(g_0, g'_0) + k_{sub}(g_1, g'_1) + \cdots + k_{sub}(g_h, g'_h)$$

The function k_{sub} is a subtree kernel function.

Definition 4.2 (Weisfeiler-Lehman subtree kernel). The subtree graph kernel of g and g' is

$$k_{sub}(g, g') = \langle \phi(g), \phi(g') \rangle$$

where the notation $\langle ., . \rangle$ denotes the inner product of two vectors.

The ϕ function vectorizes a labeled graph by counting the original and compressed node labels of the graphs in the WL sequences of g and g'. Specifically, let Σ_i be the set of node labels that occur at least once in g or g' at the end of the i-th iteration of the algorithm, and let $c_i(g, \sigma_{ij})$ be the number of occurrences of the label $\sigma_{ij} \in \Sigma_i$ in the graph g. Based on the counter c_i, we compute ϕ as follows:

$$\phi(g) = \big(c_0(g, \sigma_{01}), \ldots, c_0(g, \sigma_{0|\Sigma_0|}), \ldots, c_h(g, \sigma_{h1}), \ldots, c_h(g, \sigma_{h|\Sigma_h|})\big)$$

4.2 Training

The goal of the training step of TSFinder is to create a binary classification model that predicts whether a given class is thread-safe or thread-unsafe. We use a supervised learning technique and therefore require training data. As training data, we use two sets of classes C_{TS} and $C_{\widetilde{TS}}$, which consist of known thread-safe and known thread-unsafe classes, respectively. For each of these classes, the static analysis (§ 3) extracts a set of graphs.

4.2.1 Graphs Vectors. TSFinder first computes a vector representation of each graph based on the graph kernel function in § 4.1. Intuitively, the vector characterizes a graph by summarizing how similar it is to other, known graphs in the training data.

More technically, the approach computes the vector representation of a graph in three steps:

(1) Fix the order of all graphs in $G_{C_{TS} \cup C_{\widetilde{TS}}}$ to obtain a list of graphs $g_1, .., g_n$. The specific order does not matter, as long as it remains fixed.
(2) Compute the kernel matrix of all graphs $K = (k(g_i, g_j))_{i,j}$ for $1 \leq i, j \leq n$.
(3) For each graph g_i, the i-th row of K is the vector representation of g, called graph vector.

4.2.2 Combining Class Graphs. Given the graphs vectors of a class, we combine these vectors into a single class vector. Intuitively, the class vector should summarize to what extent the individual graphs of a class resemble the graphs of classes in the training data. If a class has graphs that are very similar to graphs that typically occur in thread-safe classes, then the class is more likely to thread-safe. Likewise, a class with graphs that mostly resemble graphs from thread-unsafe classes is more likely to also be thread-unsafe. To encode this intuition, we create a class vector by computing the minimum, maximum, and average similarity of all the graphs of the class against all graphs extracted from the training classes.

Let $n = |G_{C_{TS} \cup C_{\widetilde{TS}}}|$ be the total number of graphs extracted from all classes in the training data. For a specific class C, let G_C be the set of all graphs TSFinder extracted from C and $m = |G_C|$ be the total number of these graphs. For each graph $g_i \in G_C$ where $1 < i < m$, let $f_{g_i}^j$ where $1 < j < n$ be the jth feature of graph g_i of the class C. Our approach computes the class vector by calculating $\forall j \in 1, \ldots, n$:

$$min(f_{g_i}^j \ \forall i \in 1, \ldots, m),$$
$$max(f_{g_i}^j \ \forall i \in 1, \ldots, m),$$
$$mean(f_{g_i}^j \ \forall i \in 1, \ldots, m)$$

and concatenating these $n * 3$ values into a single vector.

For example, the class vector in Figure 2d has 3*4860=14580 elements. The first three elements are the minimum, maximum, and mean similarity of the graphs in Figure 2b compared to the first graph in the list of graphs extracted from the training classes. The next three elements are the minimum, maximum, and mean similarity of the graphs in Figure 2b compared to the second graph extracted from the training classes, …etc.

4.2.3 Classifier. Given the class vectors and their corresponding labels $l_1, .., l_n$ that indicate whether a class c is from C_{TS} or from $C_{\widetilde{TS}}$, we finally feed the labeled vectors into a traditional vector-based classification algorithm. By default, TSFinder uses a SVM for learning the classifier. Our evaluation also considers alternative algorithms.

4.3 Classifying a New Class

Once TSFinder has learned a model, we use it to predict the thread safety of a new class. Let C_{new} be the new class for which we wish to infer its supposed behavior regarding thread safety. The approach computes a class vector of C_{new} in the same way as for training. At first, TSFinder extracts field-focused graphs from C_{new}, which yields a set $G_{C_{new}}$ of graphs. For each graph $g \in G_{C_{new}}$ the approach computes the graph vector of g by computing its graph kernel against all graphs in our training data:

$$vec(g) = (k(g, g_j))_{1j}, j = 1, 2, \ldots, n$$

where $g_j \in G_{C_{TS} \cup C_{\widetilde{TS}}}$ is the set of graphs in the learned model and $n = |G_{C_{TS} \cup C_{\widetilde{TS}}}|$ is the total number of graphs in the model. Given the set of graphs vectors, TSFinder combines these graphs into a single class vector as described in § 4.2.2 and queries the trained model to obtain a classification label for the class C_{new}. The label indicates whether the model predicts the class to be thread-safe or thread-unsafe.

5 IMPLEMENTATION

We implement TSFinder into a fully automated tool to analyze Java classes. The static analysis builds on the static analysis framework Soot [64]. Given a class, either as source code or byte code, the analysis extracts field-focused graphs by traversing all program elements, by querying the call graph, and by analyzing definition-use relationships of statements. We use the GraphML format [11] to store graphs. To compute the WL graph kernel, we build on an existing Python implementation [58]. The SVM model is implemented on top of the Weka framework [21]. Our implementation is available as open-source.[5]

6 EVALUATION

The evaluation is driven by four main research questions:

- RQ1: How many classes come with documentation about their thread safety? (§ 6.1)
- RQ2: How effective is TSFinder in classifying classes as thread-safe or thread-unsafe? (§ 6.2)
- RQ3: How efficient is TSFinder? (§ 6.3)
- RQ4: How does TSFinder compare to variants of the approach and to a simpler approach? (§ 6.4)

6.1 RQ1: Existing Thread Safety Documentation

To better understand the current state-of-the art in documenting thread safety, we systematically search all 179,239 classes from

[5] https://github.com/sola-da/TSFinder

the Qualitas corpus for thread safety documentation. We focus on documentation provided as part of the Javadoc comments of a class and its members, and ignore any other documentation, e.g., on project web sites or in books. Most real-world classes have Javadoc documentation and it is a common software engineering practice to document class-level properties, such as thread safety, there.

Our inspection proceeds in two steps. At first, we generate the Javadoc HTML files for all classes and automatically search them for keywords related to concurrency and thread safety. Specifically, we search for “concu”, “thread”, “sync”, and “parallel”. We choose these terms to overapproximate any relevant documentation. In total, the search yields hits in 8,655 of the 179,239 classes.

As the second step, we manually analyze a random sample of 120 of the 8,655 classes. For each sampled class, we inspect the Javadoc and search for any documentation related to the thread safety of the class. Based on this inspection, we classify the class in one of the following four categories.

Documented as thread-safe. The documentation explicitly specifies that the class is supposed to be thread-safe or this intention can be clearly derived from the available information. Examples include:

- The class-level documentation states “This class is thread-safe”.
- The name of the class is `SynchronousXYChart` and the project also contains a class `XYChart`, indicating that the former is a thread-safe variant of the latter.
- The class-level documentation states “Mutex that allows only one thread to proceed [while] other threads must wait until the one finishes”. The semantics of a mutex implementation imply that the class is thread-safe because mutexes are accessed concurrently without acquiring any additional locks.

Documented as thread-unsafe. The documentation explicitly specifies that the class is not supposed to be thread-safe or this intention can be clearly derived from the available information. Examples include:

- The class-level documentation states “This class is not thread-safe” or “not to be used without synchronization”.
- The class-level documentation states “We are not using any synchronized so that this does not become a bottleneck”.
- The class-level documentation states “The class (..) shall be used according to the Swing threading model”, which implies that only the Swing thread may access instances of the class and that the class is not thread-safe [77].

Documented as conditionally thread-safe. The documentation specifies the class to be thread-safe under some condition. Examples include:

- The class depends on another class and has the same thread safety as this other class.
- All static methods of the class are thread-safe, whereas non-static methods are not necessarily thread-safe.

No documentation on thread safety. The documentation does not mention thread safety and we cannot derive from other available information whether the class is supposed to be thread-safe. Examples of documentation that matches our search terms but does not document thread safety include:

- The class implements a graph data structure and its documentation says that it “permits *parallel* edges”.
- The method-level documentation specifies that an argument or the return value of the method is supposed to be thread-safe. While such a statement is about thread safety, it does not specify this property for the current class.

Table 1: Existing thread safety documentation.

Documented as:	Number	Percentage
Thread-safe	11	9.2%
Not thread-safe	12	10.0%
Conditionally thread-safe	2	1.7%
No documentation	95	79.2%
Total inspected	120	100.0%

Table 1 summarizes the results of this classification. We find that most (79.2%) of the inspected classes do not document the thread safety of the class, but hit our search terms for some other reason. In the documented subset of classes, which sums up to 20.8%, roughly the same number of classes is documented as thread-safe and thread-unsafe, respectively.

Under the assumptions that our search terms cover all possible thread safety documentation and that the 120 sampled classes are representative for the entire population of classes in the corpus, we can estimate the percentage of documented classes in the corpus:

$$\frac{\%\ documented\ *\ Search\ hits}{Total\ classes} = \frac{0.208 * 8,655}{179,239} = 1.004\%$$

> In summary, the vast majority of real-world Java classes do not document whether they are thread-safe or not. Among the few documented classes, 47.8% and 52.2% are documented as thread-safe and thread-unsafe, respectively. We conclude that the current state of thread safety documentation is poor and will benefit from automatic inference of documentation.

6.2 RQ2: Effectiveness of TSFinder

6.2.1 Data Set and Graph Extraction. For the remaining evaluation, we use a set of 230 classes gathered from JDK version 1.8.0_152. These classes are documented to be either thread-safe or thread-unsafe, providing a ground truth for our evaluation. Table 2 shows the number fo fields, methods, and lines of code of these classes. In total, the classes sum up to 74,313 lines of Java code. The last three columns of Table 2 provide statistics about the graphs that TSFinder extracts. On average, the static analysis extracts 21.1 graphs per class, which yields a total of 4,860 graphs that the approach learns from.

Although the number of thread-safe and thread-unsafe classes is equal, the total number of extracted graphs from thread-unsafe classes is about 1.4 the number of graphs extracted from thread-safe classes. Since TSFinder uses the entire set of 4,860 graphs to construct the class vector for any class, this imbalance does

Table 2: Classes and extracted field-focused graphs used for training and cross-validation.

		Fields			Methods			LoC			Extracted graphs		
Classes	Count	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Graphs	Vertices	Edges
Thread-safe	115	1	64	8.7	2	163	34.7	13	4,264	430.2	1,989	128,493	150,850
Thread-unsafe	115	0	55	4.3	1	103	23.8	7	1,931	219.7	2,871	151,410	170,473
All	230	0	64	6.4	1	163	29.2	7	4,264	323.1	4,860	279,903	321,323

Table 3: Effectiveness of classification via 10-fold cross validation across 230 classes with $h = 3$.

	Thread-safe		Thread-unsafe	
Accuracy	Precision	Recall	Precision	Recall
94.5%	94.9%	94.0%	94.2%	95.0%

not prevent the approach from learning an effective classifier. The number of graphs per category in Table 2 is disproportionate to the number of fields and methods in the same category due to flattening the class hierarchy (§ 3.1.3).

6.2.2 Results. To evaluate the effectiveness of TSFinder, we apply it to the 230 classes and measure precision, recall, and accuracy. We perform 10-fold cross validation, a standard technique to evaluate supervised machine learning. The technique shuffles and splits all labeled data, i.e., our 230 thread-safe and thread-unsafe classes into ten equally sized sets. For each set, it then trains a model with the classes in the other nine sets and predicts the labels of the remaining classes using the trained model. We measure accuracy as the percentage of correct classifications among all classifications made by TSFinder. We measure precision and recall both for predicting thread safety and for predicting thread unsafety. With respect to thread (un)safety, precision means the percentage of correct thread (un)safety predictions among all predictions saying that a class is thread-(un)safe. Recall means the percentage of classes classified as thread-(un)safe among all classes that are actually thread-(un)safe.

Table 3 shows the results of the 10-fold cross validation. The classification accuracy is 94.5%, i.e., TSFinder correctly predicts the thread safety of the vast majority of classes. The precision and recall results allow the reader to further understand how incorrect predictions are distributed. For example, the fact that the precision for thread safety is 94.9% means the following: When the approach predicts a class to be thread-safe, then this prediction is correct in 94.9% of the cases. Similar, the recall for thread-safety of 94.0% means that TSFinder finds 94.0% of all thread-safe classes and misses the remaining 6%.

6.2.3 Manual Inspection. To better understand the limitations of TSFinder, we inspect some of the mis-classified classes.

Thread-safe class predicted as not thread-safe. TSFinder mistakenly predicts the thread-safe `ConcurrentLinkedQueue` class to be thread-unsafe. This queue implementation builds upon a non-blocking algorithm [38]. Since our training set includes only six classes that use a similar lock-free implementation, the training data may not be sufficient for the classifier to generalize to the `ConcurrentLinkedQueue` implementation. Nevertheless, TSFinder correctly predicts some of the other classes that use non-blocking implementations.

Thread-unsafe class predicted as thread-safe. The approach predicts `TreeSet` and `EnumSet` as thread-safe, even though they are thread-unsafe implementations of the abstract class `AbstractSet`. We suspect these misclassification to be due limitations of the the learned model to generalize to previously unseen cases.

Inaccurate documentation. TSFinder classifies `PKIXCertPath-ValidatorResult` as thread-safe, even though its documentation labels it as not thread-safe. Manually inspecting the implementation shows that the class is indeed thread-safe. The private fields of the class are initialized by the constructor and after that cannot be written to. This case illustrates that TSFinder can not only add otherwise missing documentation, but could also be useful for validating existing documentation.

In summary, our classifier correctly predicts the thread safety of a class in 94.5% of the cases. The precision and recall for identifying thread-safe classes are 94.9% and 94.0%, respectively. We conclude that the approach achieves its goal of automatically and precisely identifying whether a class is supposed to be thread-safe.

6.3 RQ3: Efficiency of TSFinder

We evaluate the efficiency of our approach by measuring the time required for the different steps. All experiments are performed on a machine with 4 Intel i7-4600U CPU cores and 12GB of memory. Training the classifier with a set of training classes is a one-time effort. For the 230 training classes that we use in this evaluation, the training takes approximately 11.7 minutes, including all computation steps, such as extracting graphs, computing graph kernels, and training the SVM model. When querying TSFinder with a new class, the approach extracts graphs from this class and classifies the class based on the graphs. The former step takes about 3 seconds and it dominates the latter which is negligible, on average over all 230 training classes.

TSFinder stores graphs extracted from training classes as part of its trained model. These graphs are used to compute the pairwise similarity of graphs extracted from the class under inspection to build the vector embedding of the class. For our model trained with 230 graphs, the total size of the compressed graphs is 0.6 MB, i.e., the space consumed by the model graphs is negligible.

Table 4: Effect of the WL kernel iterations parameter h on classification.

h	1	2	3	4	5	6	7
Accuracy (%)	89.7	94.1	94.5	94.4	93.9	94.1	94.1

Table 5: Effectiveness of the graph-based TSFinder against the SimpleClassifier classifier.

	Accuracy	
Classifier	TSFinder	SimpleClassifier
SVM (SGD[6] with hinge loss)	94.5%	75.0%
Random forest	94.1%	79.3%
SVM (SMO[7])	92.5%	70.6%
SVM (SGD with log loss)	92.0%	74.3%
Additive logistic regression	92.8%	74.5%

We conclude that TSFinder is time and space efficient enough to document hundreds of classes, e.g., of a third-party library, in reasonable time and with minimal space overhead.

6.4 RQ4: Comparison with Alternative Approaches

As the default classification algorithm, we use a SVM with stochastic gradient decent (SGD) and the hinge loss function. We empirically set the learning rate to 0.0001 and the number of WL-iteration h to 3. The following compares this configuration with alternative approaches.

6.4.1 Configuration of the WL Graph Kernel. To compare field-focused graphs with each other, TSFinder uses the WL graph kernel, which has a parameter h that determines to what extent should it compress node labels. Table 4 shows the effect of h on the classification accuracy. The results suggest that $h = 3$ is an appropriate value for h and that small variations of the parameter do not significantly change the accuracy.

6.4.2 Classification Algorithm. TSFinder uses a classification algorithm that determines whether a given class vector is likely thread-safe or not (§ 4.2.3). We evaluate several other popular algorithms in addition to our default of SVM with stochastic gradient descent and hinge loss. Table 5 shows the accuracy of TSFinder with four other classification algorithms, each with the default configuration of hyperparameters provided by Weka. The results show that the accuracy is only slightly influenced by the choice of classification algorithm, as it ranges between 92.0% and 94.5%.

6.4.3 Simple Class-level Features. We evaluate whether our graph-based view on classes could be replaced by a simpler approach that summarizes class-level features into a vector. The intuition behind this set of features is that as a human, we tend to believe that, for example, a class with high percentage of synchronized methods is probably more likely intended to be thread-safe than a class with fewer synchronized methods. Specifically, we consider the following class-level features:

- Percentage of fields that are volatile.
- Percentage of fields that are public and volatile.
- Percentage of methods that are either synchronized or contain a synchronized block.
- Percentage of methods that are either public and synchronized or public and contain a synchronized block.

Based on a feature vector for each of our 230 classes, we train and evaluate a classifier using the same 10-fold cross validation strategy as above. We call this approach *SimpleClassifier*. The last column in Table 5 shows the accuracy obtained by SimpleClassifier using different learning algorithms. All algorithms are used with their default configurations, as provided by Weka. The highest accuracy that SimpleClassifier achieves is 79.3%, using the random forest learning algorithm, which is significantly lower than the accuracy of TSFinder.

In summary, we find that the choice of classification algorithm has little influence on the accuracy of TSFinder. Comparing the approach with a classifier based on simple, class-level features shows that our graph-based representation of classes yields a significantly more accurate classifier (94.5% versus 79.3%).

7 LIMITATIONS

One limitation is that the training classes may not comprehensively cover all possible patterns of thread-safe and thread-unsafe code. As a result, the analysis may not be able to correctly classify a previously unseen class that relies on a completely new way to implement thread safety. We try to address this problem by selecting a diverse set of training classes that are used in various application domains and that cover different concurrency-related implementation patterns, e.g., immutable classes, classes that use synchronized methods, and classes that use synchronization blocks.

Another limitation is that some of the supposedly thread-safe training classes may have subtle concurrency bugs. If such bugs were prevalent, the approach might learn patterns of buggy concurrent code. To mitigate this potential problem, the training set contains well-tested and widely used classes, for which we assume that most of their implementation is correct.

8 RELATED WORK

8.1 Analysis of Concurrent Code

The analysis of concurrent software has been an active topic for several years. Analyses that target thread-safe classes are particularly related to our work. ConTeGe [44] and Ballerina [41] have pioneered test generation for such classes. Other test generators improve upon them by considering coverage information [14, 61], by steering test generation based on sequential test executions [51–53], by comparing thread-safe classes against their superclasses [45], or by targeting tests that raise exceptions [54]. SpeedGun [46] aims at detecting performance regression bugs in thread-safe classes. ConCrash [8] creates tests that reproduce previously observed crashes.

[6] Stochastic Gradient Descent
[7] Sequential Minimal Optimization

LockPeeker [34] tests API methods to find latent locking bugs. All these approaches find correctness or performance bugs in thread-safe classes. Instead, our work addresses the orthogonal problem of inferring whether a class is even supposed to be thread-safe.

Beyond thread-safe classes, various dynamic analyses to find concurrency bugs have been proposed, such as data race detectors [13, 19, 42, 55], analyses to detect atomicity violations [7, 18, 35, 71, 78], and analyses to find other kinds of concurrency anomalies [36, 68]. While these techniques analyze a given execution, another direction is to influence the schedule of an execution to increase the chance to trigger concurrency-related misbehavior. Work on influencing schedules includes random-based scheduling [12, 15], systematic exploration of schedules [39, 66], and forcing schedules to trigger previously identified, potential bugs [28, 57]. All these approaches aim at bug detection, whereas TSFinder infers documentation.

Finally, there are various static analyses of concurrent code, e.g., to find deadlocks [4, 40, 76], atomicity violations [20], locking policies [17], and conflicting objects [69]. One strength of our work compared to existing static analyses of concurrent code is the use of a relatively simple static analysis and complementing it with graph-based machine learning.

8.2 API Documentation

Lethbridge et al. [32] study how documentation is used in practice. They find that documentation is often outdated and inconsistent. Inferring documentation from source code alleviates these problems. Another study focuses on problems that developers face when learning a new API [50]. Their results include that many APIs need more and better documentation. Our work addresses this problem by providing an automated way to generate concurrency specifications.

Improving documentation and how developers use it is an active area of research. McBurney et al. [37] investigate how to prioritize documentation effort based on source code attributes and textual analysis. Treude and Robillard [63] augment API documentation with relevant and otherwise missing information from Stack Overflow. APIBot is a bot created to answer natural language questions by developers based on the available documentation [62]. Other work finds relevant tutorial fragments for an API to help developers better understand that API [27]. Another line of work searches for mistakes in existing documentation by comparing it to the documented implementation [82] or by pinpointing comments that risk becoming inconsistent when changing identifier names in the code [49]. Our work also contributes to improving and adding otherwise missing documentation, yet we tackle the so far understudied problem of inferring concurrency-related documentation.

8.3 Specifications Mining

Specification mining automatically extracts a formal specification from source code or from programs executions. Mined specification include finite-state specifications of method calls [5, 31, 43, 75], algebraic specifications [26], temporal specifications of API usages [22, 73, 79], implicit programming rules [33], and locking disciplines [16]. One benefit of mined specifications is to use them as documentation. TSFinder can be seen as a form of specification mining. In contrast to existing techniques, our work focuses on concurrency documentation and uses machine learning to learn from known examples how to infer this specification (documentation).

8.4 Graph Kernels

Kondor and Lafferty [30] and Gärtner et al. [23] introduced graph kernels and others have been proposed since then, e.g., random-walk kernels [29], shortest-path kernels [9], and subtree kernels [48]. These graph kernels have been mainly used in bioinformatics [10], in chemoinformatics [47, 59], and in web mining [72], e.g., to find similar web pages and to analyze social networks.

Some existing work applies graph kernels to software. Wagner et al. [70] analyze process trees with graph kernels to identify malware. Another approach [6] uses Markov chains constructed from instruction traces of executables [6]. Furthermore, graph kernels have been applied to statically identify malware by applying a neighborhood hash graph kernel on call graphs [24] and by using graph edit distance on API dependency graphs [80]. Our work tackles a different problem: the lack of documentation regarding multi-threaded behavior. Another difference is the kind of information that TSFinder extracts from classes and then feeds into graph kernels. Finally, to the best of our knowledge, our experimental setup is orders of magnitude larger than any other study that utilizes graph kernels in the context of program analysis.

9 CONCLUSION

This paper addresses the understudied problem of inferring concurrency-related documentation. TSFinder is an automatic approach to infer whether a class is supposed to be thread-safe or not. Our approach is a novel combination of lightweight static analysis and graph-based classification. We show that our classifier has an accuracy of 94.5% and therefore provides high-confidence documentation, while being efficient enough to scale to hundreds of classes, e.g., in a third-party library.

We envision the long-term impact of this work to be twofold. First, developers of concurrent software can use our approach to decide if and how to use third-party classes. Second, we believe that the technical contribution of this paper – combining lightweight static analysis and graph-based classification – generalizes to other problems. For example, future work could adapt the idea to other class-level properties, such as immutability, and to other code properties, such as whether a piece of code suffers from a particular kind of bug.

ACKNOWLEDGMENTS

This work was supported in part by the German Research Foundation within the Emmy Noether project ConcSys and the Perf4JS project, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, and by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

REFERENCES

- [1] [n. d.]. JBoss Platform issue 1416472. https://bugzilla.redhat.com/show_bug.cgi?id=1416472.
- [2] [n. d.]. NX issue 239. https://track.radensolutions.com/issue/NX-239.
- [3] [n. d.]. RESTEasy issue 1669. https://issues.jboss.org/browse/RESTEASY-1669.
- [4] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. 2005. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *Haifa Verification Conference*, Vol. 3875. Springer, 191–207.
- [5] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 4–16.
- [6] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. [n. d.]. Graph-based malware detection using dynamic analysis. 7, 4 ([n. d.]), 247–258. https://doi.org/10.1007/s11416-011-0152-x
- [7] Cyrille Artho, Klaus Havelund, and Armin Biere. 2003. High-level data races. *Software Testing, Verification and Reliability* 13, 4 (2003), 207–227.
- [8] Francesco A. Bianchi, Mauro Pezze, and Valerio Terragni. 2017. Reproducing Concurrency Failures from Crash Stacks. In *FSE*.
- [9] Karsten M. Borgwardt and Hans-Peter Kriegel. 2005. Shortest-Path Kernels on Graphs. In *Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM '05)*. IEEE Computer Society, Washington, DC, USA, 74–81. https://doi.org/10.1109/ICDM.2005.132
- [10] Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alex J. Smola, and Hans-Peter Kriegel. 2005. Protein Function Prediction via Graph Kernels. *Bioinformatics* 21, 1 (Jan. 2005), 47–56. https://doi.org/10.1093/bioinformatics/bti1007
- [11] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. 2002. *GraphML Progress Report Structural Layer Proposal*. Springer Berlin Heidelberg, Berlin, Heidelberg, 501–512. https://doi.org/10.1007/3-540-45848-4_59
- [12] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 167–178.
- [13] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *Conference on Programming Language Design and Implementation (PLDI)*. 258–269.
- [14] Ankit Choudhary, Shan Lu, and Michael Pradel. 2017. Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests. In *International Conference on Software Engineering (ICSE)*.
- [15] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. 2002. Multithreaded Java program test generation. *IBM Systems Journal* 41, 1 (2002), 111–125.
- [16] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier Thaine. 2016. Locking discipline inference and checking. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 1133–1144.
- [17] Cormac Flanagan and Stephen N. Freund. 2000. Type-based race detection for Java. 219–232.
- [18] Cormac Flanagan and Stephen N. Freund. 2004. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 256–267.
- [19] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 121–133.
- [20] Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. ACM, 338–349.
- [21] Eibe Frank, Mark A. Hall, Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. 2005. WEKA - A Machine Learning Workbench for Data Mining. In *The Data Mining and Knowledge Discovery Handbook*. 1305–1314.
- [22] Mark Gabel and Zhendong Su. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Symposium on Foundations of Software Engineering (FSE)*. ACM, 339–349.
- [23] Thomas Gärtner, Peter Flach, and Stefan Wrobel. 2003. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*. Springer, 129–143.
- [24] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. [n. d.]. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security* (2013) *(AISec '13)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/2517312.2517315
- [25] Zaïd Harchaoui and Francis Bach. 2007. Image classification with segmentation graph kernels. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 1–8.
- [26] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. 2007. Discovering Documentation for Java Container Classes. *IEEE Transactions on Software Engineering* 33, 8 (2007), 526–543.
- [27] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. 2017. An Unsupervised Approach for Discovering Relevant Tutorial Fragments for APIs. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 38–48. https://doi.org/10.1109/ICSE.2017.12
- [28] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Conference on Computer Aided Verification*. Springer, 675–681.
- [29] Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. 2003. Marginalized Kernels Between Labeled Graphs. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning (ICML'03)*. AAAI Press, 321–328. http://dl.acm.org/citation.cfm?id=3041838.3041879
- [30] Risi Imre Kondor and John D. Lafferty. 2002. Diffusion Kernels on Graphs and Other Discrete Input Spaces. In *Proceedings of the Nineteenth International Conference on Machine Learning (ICML '02)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 315–322. http://dl.acm.org/citation.cfm?id=645531.655996
- [31] Choonghwan Lee, Feng Chen, and Grigore Rosu. 2011. Mining Parametric Specifications. In *International Conference on Software Engineering (ICSE)*. 591–600.
- [32] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. 2003. How Software Engineers Use Documentation: The State of the Practice. *IEEE Software* 20, 6 (2003), 35–39.
- [33] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 306–315.
- [34] Z. Lin, H. Zhong, Y. Chen, and J. Zhao. 2016. LockPeeker: Detecting latent locks in Java APIs. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 368–378.
- [35] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: detecting atomicity violations via access interleaving invariants. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 37–48.
- [36] Brandon Lucia and Luis Ceze. 2009. Finding concurrency bugs with context-aware communication graphs. In *Symposium on Microarchitecture (MICRO)*. ACM, 553–563.
- [37] P. W. McBurney, S. Jiang, M. Kessentini, N. A. Kraft, A. Armaly, M. W. Mkaouer, and C. McMillan. 2017. Towards Prioritizing Documentation Effort. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2716950
- [38] Maged M Michael and Michael L Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 267–275.
- [39] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Symposium on Operating Systems Design and Implementation*. USENIX, 267–280.
- [40] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective static deadlock detection. In *International Conference on Software Engineering (ICSE)*. IEEE, 386–396.
- [41] Adrian Nistor, Qingzhou Luo, Michael Pradel, Thomas R. Gross, and Darko Marinov. 2012. BALLERINA: Automatic Generation and Clustering of Efficient Random Unit Tests for Multithreaded Code. In *International Conference on Software Engineering (ICSE)*. 727–737.
- [42] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 167–178.
- [43] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *International Conference on Automated Software Engineering (ASE)*. 371–382.
- [44] Michael Pradel and Thomas R. Gross. 2012. Fully Automatic and Precise Detection of Thread Safety Violations. In *Conference on Programming Language Design and Implementation (PLDI)*. 521–530.
- [45] Michael Pradel and Thomas R. Gross. 2013. Automatic Testing of Sequential and Concurrent Substitutability. In *International Conference on Software Engineering (ICSE)*. 282–291.
- [46] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance Regression Testing of Concurrent Classes. In *International Symposium on Software Testing and Analysis (ISSTA)*. 13–25.
- [47] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. 2005. Graph kernels for chemical informatics. *Neural networks* 18, 8 (2005), 1093–1110.
- [48] Jan Ramon and Thomas Gärtner. 2003. Expressivity versus efficiency of graph kernels. In *Proceedings of the first international workshop on mining graphs, trees and sequences*. 65–74.
- [49] Inderjot Kaur Ratol and Martin P. Robillard. 2017. Detecting fragile comments. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 112–122.
- [50] Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software* 26, 6 (2009), 27–34.
- [51] Malavika Samak and Murali Krishna Ramanathan. 2014. Multithreaded Test Synthesis for Deadlock Detection. In *Conference on Object-Oriented Programming*

Systems, Languages and Applications (OOPSLA). 473–489.

[52] Malavika Samak and Murali Krishna Ramanathan. 2015. Synthesizing tests for detecting atomicity violations. In *ESEC/FSE*. 131–142.

[53] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. 2015. Synthesizing racy tests.. In *PLDI*. 175–185.

[54] Malavika Samak, Omer Tripp, and Murali Krishna Ramanathan. 2016. Directed synthesis of failing concurrent executions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 430–446.

[55] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* 15, 4 (1997), 391–411.

[56] Bernhard Schölkopf and Alexander J Smola. 2002. *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. the MIT Press.

[57] Koushik Sen. 2008. Race directed random testing of concurrent programs. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 11–21.

[58] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. 2011. Weisfeiler-Lehman Graph Kernels. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2539–2561. http://dl.acm.org/citation.cfm?id=1953048.2078187

[59] S Joshua Swamidass, Jonathan Chen, Jocelyne Bruand, Peter Phung, Liva Ralaivola, and Pierre Baldi. 2005. Kernels for small molecules and the prediction of mutagenicity, toxicity and anti-cancer activity. *Bioinformatics* 21, suppl 1 (2005), i359–i368.

[60] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Asia Pacific Software Engineering Conference (APSEC)*.

[61] Valerio Terragni and Shing-Chi Cheung. 2016. Coverage-Driven Test Code Generation for Concurrent Classes. In *ICSE*.

[62] Yuan Tian, Ferdian Thung, Abhishek Sharma, and David Lo. 2017. APIBot: question answering bot for API documentation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 153–158.

[63] Christoph Treude and Martin P. Robillard. 2016. Augmenting API Documentation with Insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 392–403. https://doi.org/10.1145/2884781.2884800

[64] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM, 125–135.

[65] S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. [n. d.]. Graph Kernels. 11 ([n. d.]), 1201–1242. http://dl.acm.org/citation.cfm?id=1756006.1859891

[66] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engineering* 10, 2 (2003), 203–232.

[67] Bimal Viswanath, M. Ahmad Bashir, Mark Crovella, Saikat Guha, Krishna P. Gummadi, Balachander Krishnamurthy, and Alan Mislove. 2014. Towards Detecting Anomalous User Behavior in Online Social Networks.. In *USENIX Security*. 223–238.

[68] Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. In *Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 70–82.

[69] Christoph von Praun and Thomas R. Gross. 2003. Static conflict analysis for multithreaded object-oriented programs. In *Conference on Programming Languages Design and Implementation*. ACM, 115–128.

[70] C. Wagner, G. Wagener, R. State, and T. Engel. [n. d.]. Malware analysis with graph kernels and support vector machines. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)* (2009-10). 63–68. https://doi.org/10.1109/MALWARE.2009.5403018

[71] Liqiang Wang and Scott D. Stoller. 2006. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Symposium on Principles and Practice of Parallel Programming, (PPOPP)*. ACM, 137–146.

[72] Takashi Washio and Hiroshi Motoda. 2003. State of the Art of Graph-based Data Mining. *SIGKDD Explor. Newsl.* 5, 1 (July 2003), 59–68. https://doi.org/10.1145/959242.959249

[73] Andrzej Wasylkowski and Andreas Zeller. 2009. Mining Temporal Specifications from Object Usage. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 295–306.

[74] Boris Weisfeiler and AA Lehman. 1968. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia* 2, 9 (1968), 12–16.

[75] John Whaley, Michael C. Martin, and Monica S. Lam. 2002. Automatic Extraction of Object-Oriented Component Interfaces. In *Symposium on Software Testing and Analysis (ISSTA)*. ACM, 218–228.

[76] Amy Williams, William Thies, and Michael D. Ernst. 2005. Static Deadlock Detection for Java Libraries. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 602–629.

[77] Henry Wong and Scott Oaks. 2004. *Java Threads* (3rd edition ed.). O'Reilly Media, Inc.

[78] Min Xu, Rastislav Bodík, and Mark D. Hill. 2005. A serializability violation detector for shared-memory server programs. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1–14.

[79] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining temporal API rules from imperfect traces. In *International Conference on Software Engineering (ICSE)*. ACM, 282–291.

[80] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. [n. d.]. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014) *(CCS '14)*. ACM, New York, NY, USA, 1105–1116. https://doi.org/10.1145/2660267.2660359

[81] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 27–37. https://doi.org/10.1109/ICSE.2017.11

[82] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald C. Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 27–37. https://doi.org/10.1109/ICSE.2017.11