Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

11-2016

Towards using concurrent Java API correctly

Shuang LIU

Guangdong BAI

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Jin Song DONG

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Citation

LIU, Shuang; BAI, Guangdong; SUN, Jun; and DONG, Jin Song. Towards using concurrent Java API correctly. (2016). *Proceedings if the 21st International Conference on Engineering of Complex Computer Systems, Dubai, United Arab Emirates, November 6-8*. 219-222.
Available at: https://ink.library.smu.edu.sg/sis_research/4942

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

2016 21st International Conference on Engineering of Complex Computer Systems

Towards Using Concurrent Java API Correctly

Shuang Liu*,Guangdong Bai*, Jun Sun‡,and Jin Song Dong†
*Singapore Institute of Technology †National University of Singapore
‡Singapore University of Technology and Design

***Abstract*—Concurrent Programs are hard to analyze or debug due to the complex program logic and unpredictable execution environment. In practice, ordinary programmers often adopt existing well-designed concurrency related API (e.g., those in *java.util.concurrent*) so as to avoid dealing with these issues. These API can however often be used incorrectly, which results in hard-to-debug concurrent bugs. In this work, we propose an approach for enforcing the correct usage of concurrency-related Java API. Our idea is to annotate concurrency-related Java classes with annotations related to misuse of these API and develop lightweight type checker to detect concurrent API misuse based on the annotations. To automate this process, we need to solve two problems: (1) how do we obtain annotations of the relevant API; and (2) how do we systematically detect concurrent API misuse based on the annotations? We solve the first problem by extracting annotations from the API documentation using natural language processing techniques. We solve the second problem by implementing our type checkers in the Checker Framework to detect concurrent API misuse. We apply our approach to extract annotations for all classes in the Java standard library and use them to detect concurrent API misuse in open source projects on GitHub. We confirm that concurrent API misuse is common and often results in bugs or inefficiency.**

I. Introduction

Concurrent programs are becoming prevalent not only in large programs which run on servers, but also in programs which run on laptops or even mobile phones due to the rapid advancement and wide adoption of multi-core hardware. Concurrency related issues like bugs or inefficiency are difficult to analyze due to the complex program logic and non-deterministic interleaving orders.

As discovered by Lu *et al.* [14], wrong assumptions on synchronization/ordering intentions are usually responsible for many concurrency bugs. Furthermore, we often observe that even when programmers have correct intentions of synchronization, they may still write programs with concurrency issues due to the lack of understanding of certain concurrency-related API. The Java API specification [2] is the official specification for Java standard edition maintained by Oracle. The Java API specification is written mainly in natural language. It provides detailed descriptions on how to use existing Java API, including usage rules that should be obeyed and properties that should be preserved, for correct usage of the API. However, programmers usually do not read the specifications carefully or misinterpret some of the concepts described in the specification. As a result, they may use those APIs incorrectly, which may potentially cause bugs or inefficiency in their programs.

Java provides a concurrency-related library which provides commonly used utilities and data structures. The library contains thread-safe implementations for many data structures. For instance, the *ConcurrentHashMap* class implements a hash table which supports full concurrency for retrieval operations and high concurrency for update operations. It allows multiple threads accessing concurrently but does not entail exclusive access. Therefore, when exclusive access is required on *ConcurrentHashMap* , properly designed additional locking policy should be in place.

The code snippet in Figure 1 is taken from the *Indic Keyboard* project [1], which is a versatile keyboard for Android users. The *sLangUserHistoryDictCache* field is an instance of *ConcurrentHashMap* . There are two methods, i.e., *getUserHistoryDictionary* and *runGCOnAllDictionariesIfRequired*, accessing this field. However, in method *runGCOnAllDictionariesIfRequired*, there is no explicit synchronization on *sLangUserHistoryDictCache* when it is updated (line 25). As a result, there is a potential data race. Suppose *Thread1* is created to execute method *getUserHistoryDictionary* and *Thread2* is created to execute method *RunCOnAllOpenedUserHistoryDictionaries*. *Thread1* is scheduled to execute first and it checks that the *ConcurrentHashMap* contains the key *localeStr* (line 9), then *Thread2* is scheduled to run and it invokes the *runGCOnAllDictionariesIfRequired* with the shared variable *sLangUserHistoryDictCache*. Since the code run by *Thread2* does not synchronize on the *sLangUserHistoryDictCache* object, it can access (in this case *remove*) an entry from the object. Suppose the entry is exactly the one indicated by *localeStr*. When *Thread1* resumes and runs to line 10, a *null* value is returned which violates the program logic and results in *NullPointerException* later.

This result is due to the lack of proper synchronization on the shared variable *sLangUserHistoryDictCache* when exclusive access is required (line 20-25). This may be because of the misunderstanding of the *ConcurrentHashMap* class, which enables concurrent access but does not guarantee exclusive access. *ConcurrentHashMap* uses private locks internally and therefore line 10 and line 25 would lock on different objects and as a result it is as good as no locking at all. A fix of this issue would be to guard line 25 with the same lock (i.e., *sLangUserHistoryDictCache*) as line 10. This bug may be hard to find also because the shared variable *sLangUserHistoryDictCache* is passed to the method *runGCOnAllDictionariesIfRequired* as a parameter, which has a different name, i.e., *dictionaryMap*. Thus it is easier for programmers to ignore the concurrent access to the shared variable. The trouble with synchronizing on *ConcurrentHashMap* is that it would result in inefficiency, as in such a case, all "smart" locking mechanism done inside *ConcurrentHashMap* (called lock stripping) is wasted and it would lead to lock contention on object *sLangUserHistoryDictCache*.

Misunderstanding of API specification can cause issues

978-1-5090-5527-2/16 $31.00 © 2016 IEEE
DOI 10.1109/ICECCS.2016.32

```java
public class PersonalizationHelper {
  final String localeStr = locale.toString();
  private static final ConcurrentHashMap<String, SoftReference<UserHistoryDictionary>>
    sLangUserHistoryDictCache = new ConcurrentHashMap<>();
  ...
  public static UserHistoryDictionary getUserHistoryDictionary(final Context context, final Locale locale) {
     ...
   synchronized (sLangUserHistoryDictCache) {
     if (sLangUserHistoryDictCache.containsKey(localeStr)) {
       final SoftReference<UserHistoryDictionary> ref = sLangUserHistoryDictCache.get(localeStr);
       ...
     }
  }//end synchronized
  ...
} //end method
public static void runGCOnAllOpenedUserHistoryDictionaries() {
  runGCOnAllDictionariesIfRequired(sLangUserHistoryDictCache);
}
private static <T extends DecayingExpandableBinaryDictionaryBase> void runGCOnAllDictionariesIfRequired(final
      ConcurrentHashMap<String, SoftReference<T>> dictionaryMap) {
 for (final ConcurrentHashMap.Entry<String, SoftReference<T>> entry: dictionaryMap.entrySet()) {
   final DecayingExpandableBinaryDictionaryBase dict = entry.getValue().get();
   if (dict != null) {
     dict.runGCIfRequired();
   } else {
     dictionaryMap.remove(entry.getKey());
   }
 } } }
```

Fig. 1. Code Snippet from the Indic Keyboard Project Illustrating Misuse of Clientside Locking

which are hard to detect since it is assumed to be correct by the programmers. In this work, we propose an approach for enforcing the correct usage of concurrency-related Java API. Our idea is to annotate concurrency-related Java classes with annotations which are related to misuse of these APIs and then develop lightweight type checkers to automatically detect concurrent API misuse based on the annotations. To automate this process, we must solve two problems: (1) how to obtain annotations of the relevant API; and (2) how to systematically detect concurrent API misuse based on the annotations? We solve the first problem by extracting annotations from the informal natural language Java API documentation using natural language processing techniques. We solve the second problem by developing and implementing our type checkers, based on the annotations, in the Checker Framework [16] to detect concurrent API misuse automatically.

Related Work There have been many approaches which focus on finding concurrency related issues, including deadlock [9], [12], [15], [19], data race [7], [9], [20] and atomicity violation [10], [11], [13], [18], etc. A variety of techniques such as type systems [17], static analysis [9] and dynamic analysis [20] have been explored. All the above surveyed approaches try to check either violation of certain disciplines, such as the locking discipline, the happen-before discipline, etc; or the violation of existing templates. Our work differs from them as we focus on enforcing the correct usage (misuse of which may lead to concurrency bugs) of Java API in concurrent context based on its specifications. Rather than focusing on a particular kind of buggy template, we focus on finding misuse of certain API/class due to misunderstanding of the corresponding API/class.

II. Generating Annotations

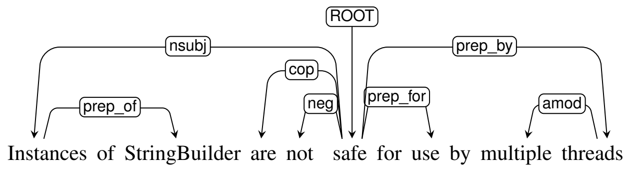
In this section, we discuss how to obtain concurrency-related annotations automatically.

Extracting Relevant Sentences We first conduct keyword matching on Java API descriptions to gather all parts of the Java API specification which may be relevant to the concurrency API specification, while filtering out those that are irrelevant. In particular, we are interested in descriptive sentences, e.g., descriptions of packages and classes in the Java API specification. We conduct standard pre-processing steps, including truncation and tokenization, to automatically regularize the sentences. We use splitta [4] to tokenize and identify sentence boundaries.

To extract the sentences which may be useful in generating the annotations, we first provide a set of seed keywords, which are semantically synonymous with the annotations that we are interested in, i.e., "reentrancy", "thread-safe", "immutable" and "locking". We obtained 530 sentences containing the keywords and they are used to extract annotations next.

Extract Annotations The second step is to extract annotations from those annotation-containing sentences. We adopt the format of typed annotation [3] to represent rules since it is a well-defined meta-data type in Java and can be used for both manual inspection and detecting errors during compiling time. Since the Java API specification is written mainly in natural language, we adopt NLP techniques, i.e., dependency parsing, to parse the sentences and extract annotations.

(1) Natural Language Parsing In this work, we adopt the Stanford Parser [5] for dependency parsing [8]. In dependency parsing, a dependency relation is associated to each pair of words in a sentence. The parsing result is usually a dependency tree, which captures the grammatical relation between words in the sentence. For example, in Figure 2, the labels on the directed arcs are Stanford dependency labels. The label *nsubj* from the word *safe* to the word *Instances* captures the grammatical relation that the word *Instances* is the nominal subject of the word *safe*. We parse the sentences from the Java API specification and obtain one dependency tree for


ROOT
nsubj
cop
prep_by
neg
prep_for
prep_of
amod
Instances of StringBuilder are not safe for use by multiple threads
Fig. 2. Example of a dependency tree

each sentence.

(2) Extracting Annotations We provide a set of rules to extract annotations from the dependency trees based mainly on the dependency relations. The format of our extracted annotation is "@AnnotationType(ObjectName)", where "AnnotationType" indicates the type of the annotation and an "ObjectName" indicates the class/object the annotation keyword restricts on.

To extract annotations in the above-mentioned format, we focus on two kinds of information, i.e., the subject and the main verb of a sentence, when analyzing the parse trees. The main verb of a sentence potentially identifies the annotation keyword and the subject of a sentence maps to the class, object or method that the corresponding annotation keyword restricts on. In this work, we are interested in four annotations, i.e., @threadsafe, @immutable, @entailLocking and @reentrant, and also their negations like @!threadsafe, which means that the class is not thread-safe. Therefore the corresponding keywords "thread safe", "immutable", "reentrant" and "exclusive, lock" as well as their synonyms are used in our rules as keywords to identify annotations from the sentences. The rules are proposed based on general English grammar and the heuristics that we discovered during the manual inspection of the sentences extracted in the previous step. The general idea is that the main verb should map to one of the annotation keywords. The subject of the sentence should correspond to the object/class/method names that the annotation keywords restrict on. Due to space limitations, please refer to our website [6] for details of the rules to extract annotations.

Evaluation We apply the above method to the sentences we obtained from Java specification. In the following, we evaluate the accuracy of the automatically extracted annotations through manual inspection. We obtained in total 530 sentences through keyword matching, 447 of which are confirmed to be annotation-containing. We extracted in total 410 annotations (340 are distinct) from the sentences. Note that One sentence may contain multiple annotations. Different sentences may also contain the same annotation. In 101 cases, we fail to extract the correct annotations. The reasons for not being able to extract correct annotations include: (1) *Parsing error due to the Stanford Parser*: The POS tags of the ROOT are incorrectly labeled, and thus leads to the incorrect results. (2) *Keywords appeared in a subclause*: Because our matching rules are based on dependency labels, which concentrate on the main clause. The annotation containing clauses which appear as subclauses are not captured. (3) *Sentences which have very complex structure/logic*: For example, "*If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.*". Complex semantic information is needed in order to understand the sentence and extract the annotation "*@!threadsafe(ArrayList)*" from it.

The full list of classes with annotations can be found at [6]. We remark that we observe that the Java specification is not complete. For instance, it is never mentioned anywhere that Boolean is immutable. As a result, our annotations are incomplete as well. Nonetheless, we believe that our automatically generated annotations would provide a good start to systematically document important concurrency related semantic information. Furthermore, our technique is not restricted to the Java specification only.

III. Using Annotations

In this section, we show how the annotations can be used to to detect potential concurrency related issues automatically.

Client-side Locking Recall that if instead private locks are used to guard the state of an (thread-safe) object, the object does not entail client side locking. We have shown misuse of those classes in Figure 1. Our approach is to develop a dedicated type checker called *locker-checker* based on the Checker Framework [16], which supports pluggable type checking for Java programs. It provides type inference utilities which enable user-defined types to be checked. To develop the type checker, we first define a type hierarchy of *locker-checker*, as shown below.

$$\begin{tikzcd}
 & \boxed{\text{@unknownLocking}} & \\
\boxed{\text{@entailLocking}} \arrow[ur] & & \boxed{\text{@!entailLocking}} \arrow[ul]
\end{tikzcd}$$

The top of the type hierarchy is type @unknownLocking, which is the default type for non-annotated objects. @entailLocking and @!entailLocking are subtypes of @unknownLocking. Objects annotated with @entailLocking are objects which guarantee exclusive access. Objects annotated with @!entailLocking are thread-safe objects which do not guarantee exclusive access.

Based on the above type hierarchy, *locker-checker* is designed to check for API misuse associated with types which are annotated with @!entailLocking. In particular, *locker-checker* is based on the following hypothesis. When a compound operation on an object annotated with @!entailLocking is contained in a synchronized block, e.g., line 8 to 13 in Figure 1, we assume that the requirement is that the compound action must be carried out atomically and thus any operation in other executing threads which may modify the object in between must be synchronized with the same lock. According to the assumption, line 25 in Figure 1 must be put in a synchronized block on *sLangUserHistoryDictCache*. Due to space limitation, we put the algorithm encoded in our locker-checker on our website [6].

Checking Locking on Immutable Objects There are potentially subtle concurrency bugs if we synchronize on an immutable object and then modify the object in certain way. To check such kind of issues, we modify the Javari [17] type checker, which is a built-in type checker in the Checker Framework, and encode rules to detect *locking-on-immutable-object* errors. We add an annotation @Immutable to the existing type hierarchy of Javari Checker and make it a subtype of @ReadOnly. To check whether the case of *lock-on-immutable-object* may happen, we add a checking rule for the @Immutable annotation to monitor whether the annotated object may be modified within a synchronized block.

Evaluation We answer two questions in the evaluation.

(1) Are there concurrent API misuses in real-world projects? The question can be answered only if we can check a large number of real-world projects. We downloaded the top 1K most popular projects from GitHub, among which 344 projects used the concurrent collection API. Among the 344 projects, 193 projects try to use the synchronized keywords on those concurrent API for exclusive access. Our method detected that 30 of those projects contain concurrency issues, meaning that 15% of the time when developers want to guarantee exclusive access on the concurrent collections, they use it wrongly.

We detected 87 misuses from 30 different projects. We manually inspect each one of them and confirm that most of them are highly likely actual bugs (e.g., there are data races). The result evidences that this kind of concurrency issues are not uncommon in those projects. The issue-containing projects vary from large platforms or servers (e.g., alibaba/dubbo, Openfire), android apps (e.g., Indic-Keyboard, MozStumbler), to popular open source libraries like GeoServer.

(2) Is our issue detection method relatively sound? To answer this question, we manually inspected all the issues reported by our method and decide whether they are real issues (bugs or inefficiency) or false alarms.

We detected 12 immutability related data races, among which 8 are manually confirmed to be potential bugs. The reason for the false alarms is that even though there is a data race, the only race is on the shared immutable variable which is assigned to a constant value (e.g., $true$). These false alarms can be avoided by improving our method to take into account special constants. We found 75 issues related to the @!entaillocking annotation, 66 of the detected problems are manually confirmed to be potential bugs.

Obviously, we would not know for sure whether a misuse is a bug as we do not know the specification of those projects. *We thus contacted the authors of the relevant projects (30 of them) and reported the issues we found in their GitHub project forums to confirm whether our findings are indeed bugs.* We got feedback from 12 of projects, which is reasonable considering not all projects are active. Among the 12, 8 of them confirmed that indeed what we found are concurrency related issues in their code. The bugs are subsequently fixed. The rest explained those are not considered bugs either because the race is considered benign; or there are other constraints in the system which makes it impossible for multiple threads to execute the two racing parts of the program at the same time. The full list of issues can be found on our website [6].

IV. Conclusion

We propose to enforce correct usage of Java concurrent API through natural language processing and type checking. We first extract concurrency related annotations from Java API document. Then we use those annotations to detect violations of API usage through type checking. We show how to detect concurrency bugs related to two types of under-explored annotations, i.e., @Immutable and @!entailLocking. We detected potential concurrency issues from 30 GitHub projects. We reported those issues to the developers and got responses from developers of 12 projects, in which 8 of them confirm to have our reported concurrency issues.

References

[1] *The Indic Keyboard Project, https://github.com/smc/Indic-Keyboard.*

[2] *Java Platform, Standard Edition 8 API Specification, https://docs.oracle.com/javase/8/docs/api/.*

[3] *Java Typed Annotation and Pluggable Type Systems Specification, https://docs.oracle.com/javase/tutorial/java/annotations.*

[4] *Splitta: a statistical sentence boundary detection tool, https://code.google.com/p/splitta/.*

[5] *The Stanford Parser, http://nlp.stanford.edu/software/lex-parser.shtml.*

[6] http://sav.sutd.edu.sg/?page_id=2845.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: proportional detection of data races. *ACM Sigplan Notices*, 45(6):255–268, 2010.

[8] M.-C. De Marneffe, B. MacCartney, C. D. Manning, et al. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, volume 6, pages 449–454, 2006.

[9] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, USA, 2003. ACM.

[10] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 256–267, New York, USA, 2004. ACM.

[11] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *In Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, pages 175–190. Springer, 2003.

[12] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 295–308, Berkeley, CA, USA, 2008. USENIX Association.

[13] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 235–244, New York, USA, 2010. ACM.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, USA, 2008. ACM.

[15] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, pages 386–396. IEEE Computer Society, 2009.

[16] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, USA, 2008. ACM.

[17] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 211–230, New York, USA, 2005. ACM.

[18] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 334–345, New York, USA, 2006. ACM.

[19] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 281–294, Berkeley, CA, USA, 2008. USENIX Association.

[20] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 221–234, New York, USA, 2005. ACM.