AUTOMATED REFACTORING FOR JAVA CONCURRENCY

BY

YU LIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

ABSTRACT

In multicore era, programmers exploit concurrent programming to gain performance and responsiveness benefits. However, concurrent programs are difficult to write: the programmer has to balance two conflicting forces, thread safety and performance.

To make concurrent programming easier, modern programming languages provide many kinds of concurrent constructs, such as threads, asynchronous tasks, concurrent collections, etc. However, despite the existence of these concurrent constructs, we know little about how developers use them. On the other hand, although existing API documentation teach developers how to use concurrent constructs, developers can still misuse and underuse them.

In this dissertation, we study the use, misuse, and underuse of two types of commonly used Java concurrent constructs: Java concurrent collections and Android async constructs. Our studies show that even though concurrent constructs are widely used in practice, developers still misuse and underuse them, causing semantic and performance bugs.

We propose and develop a refactoring toolset to help developers correctly use concurrent constructs. The toolset is composed of three automated refactorings: (i) detecting and fixing the misuses of Java concurrent collections, (ii) retrofitting concurrency for existing sequential Android code via a basic Android async construct, and (iii) converting inappropriately used basic Android async constructs to appropriately enhanced constructs for Android apps. Refactorings (i) and (iii) aim to fix misused constructs while refactoring (ii) aims to eliminate underuses.

First, we cataloged nine commonly misused CHECK-THEN-ACT idioms of Java concurrent collections, and show the correct usage of each idiom. We implemented the detection strategies in a tool, CTADETECTOR, that finds and fixes misused CHECK-THEN-ACT idioms. We applied CTADETECTOR to 28 widely used open source Java projects (comprising 6.4 million lines of code) that use Java concurrent collections. CTADetector discovered and fixed 60 bugs. These bugs were confirmed by developers and the fixes were accepted.

Second, we conducted a formative study on how a basic Android async construct, `AsyncTask`, is used, misused, and underused in Android apps. Based on the study, we designed, developed, and evaluated Asynchronizer, an automated refactoring tool that enables developers to retrofit concurrency into Android apps. The refactoring uses a points-to static analysis to determine the safety of the refactoring. We applied Asynchronizer to perform 123 refactorings in 19 widely used Android apps; their developers accepted 40 refactorings in 7 projects.

Third, we conducted a formative study on a corpus of 611 widely-used Android apps to map the asynchronous landscape of Android apps, understand how developers retrofit concurrency in Android apps, and learn about barriers encountered by developers. Based on this study, we designed, implemented, and evaluated AsyncDroid, a refactoring tool which enables Android developers to transform existing improperly-used async constructs into correct constructs. We submitted 45 refactoring patches generated by AsyncDroid in 7 widely used Android projects, and developers accepted 15 of them.

Finally, we released all tools as open-source plugins for the widely used Eclipse IDE which has millions of Java users. Moreover, we also integrated CTADetector and AsyncDroid with a static analysis platform, ShipShape, that is developed by Google. Google envisions ShipShape to become a widely-used platform. Any app developer that wants to check code quality, for example before submitting an app to the app store, would run ShipShape on her code base. We expect that by contributing new async analyzers to ShipShape, millions of app developers would benefit by being able to execute our analysis and transformations on their code.

To my parents, to my wife Jie and our coming child.

ACKNOWLEDGMENTS

I would like to thank my adviser, Danny Dig, for his support and guidance during my Ph.D study. He provided interesting directions for me to explore, and he also gave me the freedom to investigate my own ideas in this dissertation. He was patient with my research progress and always provided useful feedback timely. Without his continuous support, I would not have been able to accomplish my dissertation.

I would also like to thank Darko Marinov for his supervision during my Ph.D. program. I have been working closely with Darko in the first year of my Ph.D. study. He taught me how to be a professional researcher and engineer in computer science. Although he is not the primary guide of my dissertation, he monitored my progress and provided valuable advice.

My sincere gratitude also goes to the rest of my dissertation committee members Tao Xie and Franjo Ivancic. They provided insightful comments on the earlier drafts of my dissertation, and helped me to stay in a focused direction and have a right plan for my research.

I am thankful to Cosmin Radoi for his help with building a static data race detector for Android apps; Semih Okur for the collaboration on conducting a formative study; my dear friend and officemate Milos Gligoric for his help with exploring research ideas and preparing my qualification and preliminary exams.

I would also like to thank my Google colleagues Niranjan Tulpule and Eduardo Bravo for their comments on my dissertation topic. My thanks also go to my other colleagues and collaborators, Vilas Jagannath, Samira Tasharofi, Matt Kirn, Ralph Jonhson, Aarti Gupta, Pallavi Joshi, Gogul Balakrishnan, Malay Ganai.

I am grateful to Stas Negara, Caius Brindescu, Mihai Codoban, Michael Hilton, Alex Gyori, August Shi, Farah Hariri, Owolabi Legunsen, Sruti Srinivasa, Sergey Shmarkatyuk, Chris Scaffidi, Kendall Bailey, Shane McKee, Shanxiang Qi, Wonsun Ahn and anonymous reviewers for their feedback on the drafts of the papers that contributed to this dissertation.

During the work on my dissertation, I was partially supported by the National Science Foundation award number NSF CCF-1213091, CCF-1219027, CCF-1439957 and CCF-1442157, a SEIF award from Microsoft, a gift grant from Intel, and the Intel-Illinois Center for Parallelism at the University of Illinois at Urbana-Champaign.

Last but certainly not least, my thanks go to my family, especially my parents and my wife for their endless love and support. My parents not only provided financial support for my Ph.D. study, but also encouraged me during my thesis writing. Most of all, I deeply thank my wife who has been my best friend in my life. It is her tremendous love that encourages me to be positive and brave, and supports me to finish my Ph.D. study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The hardware industry keeps up with Moores law by resorting to multicore processing. Nowadays multicores are everywhere: in smart phones, tablets, laptops, and desktop computers. In the multicore era, the software industry can benefit from hardware improvements if they leverage concurrent programming. However, writing concurrent programs is hard: the programmer has to balance two conflicting forces, thread-safety and performance.

The industry trend is to convert the hard problem of using concurrency into the easier problem of using concurrent libraries and constructs. For example, Microsoft provides Task Parallel Library (TPL) [1] and Collections.Concurrent (CC [2]), Intel provides the Threading Building Blocks (TBB) [3], and the Java community uses the `java.util.concurrent` (`j.u.c.`) [4] library, Android framework provides `AsyncTask` [5], `IntentService` [6] and `AsyncTaskLoader` [7].

Despite several existing books and API documentation that teach how and where to use concurrent constructs, developers still misuse and underuse them. First, for concurrent collections, which is one of the most widely-used features in concurrent libraries [8, 9, 10, 11], developers often combine several operations to express higher-level semantics such as CHECK-THEN-ACT [10] idioms. In this idiom, the code first checks a condition, and then acts based on the result of the condition. Although individual operations of a concurrent collection are thread-safe, composing two operations can lead to atomicity violation bugs. Second, for the basic async construct, `AsyncTask`, provided by Android framework, developers misuse it by either launching a task but immediately blocking it to wait for the task result, or accessing unthread-safe objects (i.e., GUI widgets) in a task. On the other hand, Android async constructs are underused since there are places where we can move synchronous code into asynchronous code to improve the responsiveness of mobile apps. Finally, although

Android provides several async constructs that developers can use, developers can still choose the inappropriate async constructs, which result in memory leaks, lost results, and wasted energy.

This dissertation studies two types of Java concurrent constructs: Java concurrent collections and Android async constructs. Based on the studies, it presents three refactorings to help programmers to correctly use Java concurrent constructs. These refactorings either fix the incorrect uses of Java concurrent constructs or introduce new uses of concurrent constructs correctly. First, we present CTADetector which uses a static analysis to detect instances of misused idiom of concurrent collections and automatically correct them via refactoring. Second, we present an automated refactoring tool, Asynchronizer, which enables developers to extract blocking sequential code into `AsyncTask` for Android apps. Third, we presents a tool, AsyncDroid, which enables Android developers to transform existing improperly-used shared-memory style Android async constructs (i.e., `AsyncTask`) into proper distributed-style constructs (i.e., `IntentService`).

1.1 Thesis Statement

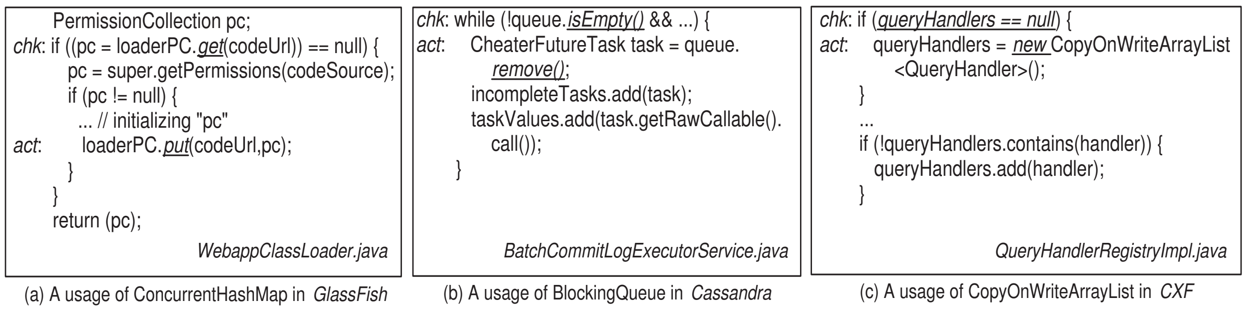
Our thesis is three-pronged:

(1) There is a need for solid studies on the uses of concurrent constructs. (2) It is possible to design and develop refactoring techniques to fix misuses and introduce new uses of concurrent constructs. (3) These refactorings are useful in practice.

To prove this statement, this dissertation presents four main bodies of research: (1) a study on Java concurrent collections and an approach for detecting and correcting check-then-act idioms of concurrent collections, (2) a study on Android `AsyncTask` and a refactoring technique for safely retrofitting `AsyncTask` to sequential code, (3) a study to map the asynchronous landscape of Android and a refactoring technique for converting shared-memory style `AsyncTask` into distributed-style communication `IntentService` for Android apps, and (4) an approach to integrate the above refactoring techniques with a single static analysis platform. For each refactoring technique, we submitted our refactoring patches to developers (a total of 323 patches) and they think our patches are useful. The rest of this chapter introduces these four bodies of research.

1.2 Check-then-Act Misuse of Java Concurrent Collections

Individual operations of concurrent collections are thread-safe (i.e., several threads can safely `put` into the same `ConcurrentHashMap` in parallel). However, concurrent collections can be easily misused. Often programmers combine several operations to express higher-level semantics such as CHECK-THEN-ACT [10] idioms. In this idiom, the code first checks a condition, and then acts based on the result of the condition.



```
      PermissionCollection pc;
chk: if ((pc = loaderPC.get(codeUrl)) == null) {
        pc = super.getPermissions(codeSource);
        if (pc != null) {
          ... // initializing "pc"
act:      loaderPC.put(codeUrl,pc);
        }
      }
      return (pc);
                          WebappClassLoader.java
```

(a) A usage of ConcurrentHashMap in *GlassFish*

```
chk: while (!queue.isEmpty() && ...) {
act:    CheaterFutureTask task = queue.
          remove();
        incompleteTasks.add(task);
        taskValues.add(task.getRawCallable().
          call());
      }
                BatchCommitLogExecutorService.java
```

(b) A usage of BlockingQueue in *Cassandra*

```
chk: if (queryHandlers == null) {
act:    queryHandlers = new CopyOnWriteArrayList
          <QueryHandler>();
      }
      ...
      if (!queryHandlers.contains(handler)) {
        queryHandlers.add(handler);
      }
                       QueryHandlerRegistryImpl.java
```

(c) A usage of CopyOnWriteArrayList in *CXF*

Figure 1.1: Three instances of misused CHECK-THEN-ACT idioms of concurrent collections used in real-world applications.

Figure 1.1 shows three real-world examples of CHECK-THEN-ACT idioms. The labels `chk` and `act` mark the check and act operations, respectively. In Fig. 1.1(a) the code checks whether a `ConcurrentHashMap loaderPC` contains a specific key and if it does not, the code creates a new value and puts it into the map. In Fig. 1.1(b) the code checks whether the queue is empty, and if not, it removes elements from the queue. Figure 1.1(c) shows a classic lazy-initialization: the code checks whether a list reference is `null`, and if so, it creates a new list and adds elements into the list.

All three examples lead to bugs when they are executed under concurrent threads, say T_1 and T_2. In Fig. 1.1(a), suppose that both T_1 and T_2 execute statement `chk` and find that the map does not contain the key. Thus, they both calculate the value and put it into the map.

Whoever is the last one will overwrite the value put by the other thread. This breaks the put-if-absent semantics of the original code. In Fig. 1.1(b), if T_2 removes the last element from the queue while T_1 is between `chk` and `act`, the element retrieved by T_1 will be null, which will lead to a `NullPointerException` in the fifth line of code. In Fig. 1.1(c) suppose that both threads find the list field is `null` and initialize it. In this case one initialization will override the other. The elements added by one thread will be lost. We found and reported all three bugs to the developers, who confirmed them as new bugs and applied our patch.

Notice that these are all examples of atomicity violation bugs: an operation executed by a thread T_2 between the T_1 thread's execution of `chk` and `act` statements might make T_1 act based on a stale condition. This can result in corrupted data structures, null pointer exceptions, and semantic errors (e.g., overwrite). Such errors can occur even if the programmers use concurrent libraries, as shown in our three examples. We call the above errors *semantic bugs*.

In addition to semantic bugs, programmers can also introduce *performance bugs* when using CHECK-THEN-ACT. A performance bug is an over-synchronized CHECK-THEN-ACT idiom that harms the performance. For the example in Fig. 1.1(a), suppose that the programmer used a lock to make the CHECK-THEN-ACT idiom atomic. However, this reduces the scalability of the application, because the same lock is used to protect all other access to the `loaderPC` map. While this correctly prevents overlapping read and write concurrent accesses, it also prevents concurrent read accesses. In an application with predominantly read accesses, the lock-based synchronization dramatically reduces the performance. A much better approach is to use the compound update APIs provided in the concurrent collections. In this example, we can change the code to use `ConcurrentHashMap.putIfAbsent`.

To address the CHECK-THEN-ACT misuses, we make the following contributions:

1. Catalog of idioms: we catalog the incorrect usage of CHECK-THEN-ACT idioms of concurrent collections.

2. Analysis of instances: By mining 28 projects, we uncover 282 misused and 545 correctly used instances of the idioms. Using this data, we answer questions about popularity, error-proneness, and evolution of idioms. This data lead to the discovery of 60 new bugs, confirmed by the developers. We reported the misused idioms that we found in this work to Doug Lea, the lead designer of `j.u.c.` package. This led them to improve some APIs in the latest JDK8 release (e.g., `compute-if-absent` in `ConcurrentHashMap`) in ways they expect will reduce the prevalence of errors and misuses.

3. Tool for detection and correction: we implemented a pattern-based static analysis tool, CTADetector, to detect misused CHECK-THEN-ACT idioms. To correct the misused idioms, our tool uses an *interactive* refactoring approach. The tool is available at: `http://refactoring.info/tools/CTADetector/`

1.3 Retrofitting Concurrency for Android Apps

For smartphone apps, responsiveness is critical. Previous research [12, 13] shows that many Android apps suffer from poor responsiveness and one of the primary reasons is that apps execute all workload in the UI event thread. The UI event thread of an Android app processes UI events, but long-running operations (i.e., CPU-bound or blocking I/O operations) will "freeze" it, so that it cannot respond to new user input.

The primary way to avoid freezing the UI event thread is to resort to concurrency, by extracting long-running operations into a background thread. Android framework provides `AsyncTask`, which is a high-level concurrent construct to execute background tasks. `AsyncTask` can also interact with the UI thread by updating the UI via event handlers. For example, the event handler `onPostExecute` executes after the task is finished, and can update the UI with the task results.

We first conducted a formative study to understand how developers use `AsyncTask`. we analyzed a corpus of top 104 most popular, open-source Android apps, comprising 1.34M SLOC, produced by 1139 developers. The study shows:

1. 48% of the studied projects use `AsyncTask` in 231 different places. In 46% of the uses, developers extracted long-running operations into `AsyncTask` via manual refactoring. Others used `AsyncTask` from the first version.

2. We found two kinds of misuses. First, in 4% of the invoked `AsyncTask`, the code runs sequentially instead of concurrently: the code launches an `AsyncTask` and immediately blocks to wait for the task's result. Second, we found that in 13 cases, code in `AsyncTask` accesses GUI widgets in a way which is not thread-safe. This leads to data races on these GUI widgets.

3. We found that 251 places in 51 projects execute long-running operations in UI event thread. This shows `AsyncTask` is underused.

Inspired by these findings, we designed, developed, and evaluated Asynchronizer, an automated refactoring tool that enables developers to use `AsyncTask`. To perform the refactoring, the developers only need to select the code that they want to run in background. Asynchronizer will automatically create an instance of `AsyncTask` as an inner class, generate event handlers in `AsyncTask`, and start the task.

However, manually applying this refactoring is non-trivial. First, a developer needs to reason about fields, method arguments, and return value for `AsyncTask`, and the statements that can be moved into `AsyncTask`. This requires reasoning about control- and data-flow. Second, the developer has to deal with several special cases. For example, `this` or `super` are relative to the enclosing class where they are used, whereas after extracting them into `AsyncTask`, they are relative to the inner `AsyncTask` class. Third, the developer needs to analyze the read-write effects on shared variables in order to prevent introducing data races.

To solve these challenges, we decompose the refactoring into two steps: code transformation and safety analysis. The transformation part uses Eclipse's refactoring engine to rewrite the code. The safety analysis uses a static race detection approach, specialized to `AsyncTask`.

This research makes the following contributions:

1. Formative study: we conducted a study on the usage, misusage, and underusage of `AsyncTask` in Android apps.

2. Algorithms: we designed the analysis and transformation algorithms to address the challenges of refactoring long-running operations into `AsyncTask`. The algorithms account for inversion of control by transforming sequential code into callback-based asynchronous code, and reason about non-interference of updates on shared variables.

3. Tool: we implemented the refactoring in a tool, Asynchronizer, integrated with the Eclipse refactoring engine. The tool is available at:

`http://refactoring.info/tools/asynchronizer/`

4. Evaluation: To evaluate Asynchronizer's usefulness, we used it to refactor 200 places in 32 open-source Android projects. We evaluate Asynchronizer from five angles. First, since 95% of the cases meet refactoring preconditions, it means that the refactoring is highly applicable. Second, in 99% of the cases, the changes applied by Asynchronizer are similar with the changes applied manually by open-source developers, thus our transformation is accurate. Third, Asynchronizer changes 2394 LOC in 62 files in just a few seconds per refactoring. Fourth, using Asynchronizer we discovered and reported 169 data races in 10 apps. 5 replied and confirmed 62 races. This shows that the automated refactoring is safer than manual refactoring. Fifth, we also submitted patches for 123 refactorings in 19 apps. 10 replied and accepted 40 refactorings. This shows that Asynchronizer is valuable.

1.4 Converting Shared-Memory into Distributed-Style Communication for Android Apps

In addition to the basic easy-to-use `AsyncTask`, Android framework also provides two enhanced yet more complicated async constructs `IntentService` and `AsyncTaskLoader`. `AsyncTask` is designed for encapsulating short-running tasks while the other two are good choices for long-running tasks. However, as our study on a corpus of 500 Android apps shows, `AsyncTask` is the most widely used construct, dominating by a factor of 3x over the other two choices combined.

However, if improperly used, `AsyncTask` can lead to memory leaks, lost results, and wasted energy. Developers usually hold a reference to a GUI component in an `AsyncTask`, so that they can easily update GUI based on task results. However, there are several instances in which the Android system can destroy and recreate a GUI component while an `AsyncTask` is running: when a user changes the screen orientation, or navigates to another screen on the same app, switches to another app, clicks the "Home" button and navigates back, etc. If the `AsyncTask` is still running and it holds GUI reference, the destroyed GUI cannot be garbage collected, which leads to memory leaks.

On the other hand, if an `AsyncTask` that finished its job updates a GUI component that has already been destroyed and recreated, the update is sent to the destroyed GUI rather than the recreated new one, and cannot be seen by the user. Thus, the task result is lost, frustrating the user. Moreover, the device wasted its energy to execute a task whose result is never used. As pointed out by many forums [14, 15, 16], this problem is widespread and is critical for long-running tasks. `IntentService` and `AsyncTaskLoader` do not have the above limitations because they do not hold a reference to GUI and instead use a radically different mechanism to communicate with the GUI. To avoid the above problems of `AsyncTask`, developers must refactor `AsyncTask` code into enhanced async constructs such as `IntentService`.

However, manually applying this refactoring is non-trivial due to drastic changes in communication with GUI. This is a challenging problem because a developer needs to transform a shared-memory based communication (through access to the same variables) into a distributed-style (through marshaling objects on special channels). First, the developer needs to determine which objects should flow into or escape from `IntentService`. Unlike `AsyncTask`, the objects that flow into or escape from `IntentService` should be serializable. Determining this requires tracing the call graph and type hierarchy. Second, the developer needs to rewire the channels of communication. Whereas `AsyncTask` provides handy event handlers for callback communication, `IntentService` does not. `IntentService` and GUI can only communicate through sending and receiving broadcast messages. This requires developer to replace event handlers with semantic-equivalent broadcast receivers, which is tedious. Third, the developer needs to infer where to register the broadcast receivers. Registering at inappropriate places can still lead to losing task results.

We first conduct a formative study to to understand how developers use different Android async constructs. We analyzed a corpus of 611 most popular open-source Android apps, comprising 6.47M SLOC. To further put the study results in a broader context, we then surveyed 10 expert Android developers. The study shows:

1. 161 (32%) of the studied apps use at least one asynchronous programming, resulting in 1893 instances. Out of these, `AsyncTask` is the most widely used.

2. The following code evolution scenario exists: developers first convert sequential code to `AsyncTask`, and those that continue to evolve the code for better use of asynchrony refactor it further into enhanced constructs.

3. Android experts think `AsyncTask` is being overused at the expense of other enhanced async constructs, and many inexperienced Android developers do not know its problem. Experts suggest `AsyncTask` should only be considered for short-running tasks (i.e., less than a second).

Inspired by the results of our study, we designed, developed, and evaluated AsyncDroid, an automated refactoring tool that transforms shared-memory into distributed-style communication in the context of Android async programming. AsyncDroid refactors `AsyncTask` into `IntentService`.

This research makes the following contributions:

Formative study: To the best of our knowledge, this paper presents the first quantitative and qualitative study to (i) map the asynchronous landscape of Android, (ii) understand how developers retrofit asynchrony, and (iii) learn about barriers encountered by developers.

Algorithms: We designed the analysis and transformation algorithms to address the challenges of refactoring from shared-memory communication (as used in `AsyncTask`) to distributed-style communication (as used in `IntentService`). The algorithm determines the incoming and outgoing objects in/from `IntentService`, replaces event handlers with broadcast messages and receivers, and infers where to register broadcast receivers.

Tool: We implemented the algorithms in AsyncDroid, a refactoring tool built on top of Eclipse refactoring engine. The tool is available at:
`http://refactoring.info/tools/asyncdroid/`

Evaluation: We evaluated AsyncDroid empirically, by refactoring 97 `AsyncTasks` in 9 popular open-source Android projects. We evaluate AsyncDroid from three aspects. First, 45% of the `AsyncTasks` pass the refactoring preconditions, and with minor manual changes another 10% `AsyncTasks` can pass preconditions. This means the refactoring is highly applicable. Second, AsyncDroid changed 3386 SLOC in 77 files in total, determined that 148 variables flow into or escape from `IntentService`, moved 14 methods into `IntentService`, and marked 18 types as serializable. This task is very large and challenging to be performed manually, but AsyncDroid performs each refactoring in a few seconds. This shows that AsyncDroid can save developer effort. Third, we submitted 45 refactoring patches in 7 projects. 4 projects replied and considered our changes to be correct, and they accepted 15 refactorings. This shows AsyncDroid is valuable.

1.5 A Platform for Practical Impact

ShipShape [17] is a static program analysis platform from Google that allows custom analyzers to plug in through a common interface. It provides interfaces to show structured analysis results and fixes for buggy code. ShipShape is packed in a docker image [18] (which is a light weight virtual machine) and takes other docker images which contain analysis tools and implement ShipShape interfaces as input. To increase the practical impact of our refactoring toolset, we also integrated CTADetector and AsyncDroid with ShipShape by putting them into publicly available docker images. Thus, users can download and invoke our tools easily from ShipShape.

1.6 Dissertation Organization

The rest of this dissertation is organized as follows:

Chapter 2: CTADetector This chapter presents the contribution of the CTADetector technique for detecting and fixing misused check-then-act idioms for Java concurrent collections.

Chapter 3: Asynchronizer This chapter presents the contributions of the Asynchronizer technique for safely introducing asynchrony to sequential Android code via refactoring.

Chapter 4: AsyncDroid This chapter presents the contributions of the AsyncDroid technique for converting improperly-used Android async constructs into proper constructs.

Chapter 5: A Platform for Practical Impact This chapter presents the contributions of integrating our toolset with a static analysis platform, ShipShape.

Chapter 6: Related Work This chapter overviews the various bodies of work that are related to the contributions of this dissertation.

Chapter 7: Conclusion and Future Work This chapter concludes the dissertation and presents various directions for future work building upon the contributions of this dissertation.

Parts of this dissertation have been published in technical reports, conferences, and journals. In particular, Chapter 2 is described in an ICST-2013 [19] and a STVR journal paper [20], Chapter 3 in a FSE-2014 paper [21] and an ASE-2015 demo paper [22], Chapter 4 in an ASE-2015 paper [23] and an ASE-2015 demo paper [22]. These chapters have been extended and revised when writing this dissertation.

CHAPTER 2

Check-then-Act Misuse of Java Concurrent Collections

2.1 Introduction

According to previous empirical studies of concurrent library usage [8, 9, 10, 11], concurrent collections are one of the most widely-used features. Concurrent collections (e.g., `ConcurrentHashMap` from `j.u.c.`) contain thread-safe, scalable data structures. Their individual operations are thread-safe. For example, several threads can safely `put` into the same `ConcurrentHashMap` in parallel. However, as shown in Sec. 1.2, concurrent collections can be easily misused in CHECK-THEN-ACT idioms which can lead to atomicity violation bugs.

Although a lot of research has been done on detecting low-level data races [24, 25, 26, 27, 28] and high-level atomicity violations [29, 30, 9, 31], there is little research on cataloging the real causes of such bugs. Because such bugs are still common, educating the developers and providing automated support on program repair is very important. Despite the fact that CHECK-THEN-ACT idioms are prevalent when using concurrent collections, there has been no work on cataloging the incorrect usage of CHECK-THEN-ACT idioms of concurrent collections.

In this chapter, we present a detailed empirical study that answers in-depth questions about the usage of CHECK-THEN-ACT idioms on a large scale. Our corpus contains 28 widely-used open-source Java projects that use concurrent collections. These projects comprise 6.4M non-blank, non-comment source lines of code (SLOC). We implemented a tool, CTADETECTOR, which uses a static analysis approach to detect instances of misused idioms and a semi-automated transformation approach to correct them. Using this data and our tool, we answer four research questions:

RQ1: *What are the commonly used* CHECK-THEN-ACT *idioms in real-world programs?* We find that in each category of correctly used and misused idioms, there is one idiom that clearly dominates the others.

RQ2: *Which idioms are the most error-prone?* We find one single idiom, `put-if-absent`, for which the number of misused instances is larger than the number of correctly used instances.

RQ3: *Do misused idioms result in real bugs? Are our patches accepted by developers?* We find 282 misused instances (218 semantic and 65 performance-related). So far we report 155 bugs to developers, and they examine 90 of them. The developers confirm 60 of the examined buggy instances as new bugs. For these confirmed bugs, the developers accept the patches generated by CTADETECTOR. The developers claim that the remaining instances do not lead to real bugs because the buggy interleaving can not not occur in practice, or the programs are resilient to such bugs.

RQ4: *What is the evolution of programs w.r.t.* CHECK-THEN-ACT *idioms?* We find that across three major versions between 2007 to 2012, the number of both correct and incorrect usages increase. However, in the later versions, the percentage of incorrect usage decreases.

There are several implications of our findings. Programmers learn a new programming construct through both positive and negative examples. Our catalog of idioms teaches them how to use CHECK-THEN-ACT idioms correctly. Along with the hundreds of instances of idioms, it provides a tremendous educational resource. Second, library designers can use our findings to make the APIs more robust or provide better documentation. Our paper has already influenced the development of the Java's `j.u.c.` package which was extended with new APIs (e.g., `computeIfAbsent`) to fix the commonly misused idioms that we found. Third, the testing community can focus its effort to find CHECK-THEN-ACT bugs in concurrent programs. Fourth, tool builders can use the developers' responses to our bug reports (Sec. 2.3.2) to think about how to create practical tools.

2.2 Catalog of Idioms

2.2.1 Thread-safety for Collections

A class is thread-safe if it behaves correctly (i.e., it conforms to its specification) when accessed from multiple threads, regardless of the scheduling or interleaving of threads, and without imposing on the calling code any additional synchronization or other coordination [10]. We will use this definition to refer now to the thread-safety of `Collection` classes.

By default, most of the Java `Collection` classes are not thread-safe. For example, the method `java.util.HashMap.put(key,value)` performs three major operations: (i) if the key is in the map, update the corresponding entry, otherwise, (ii) increase the size of the map and (iii) add a new entry. If one thread puts a new key into a `HashMap`, but the second thread gets the size of this `HashMap` between operations (ii) and (iii), the second thread will get a wrong size because the new entry has not been put yet. This scenario violates the specification of `HashMap`: the size of a map should be equal to the number of entries in the map. Thus, `HashMap` is not thread-safe.

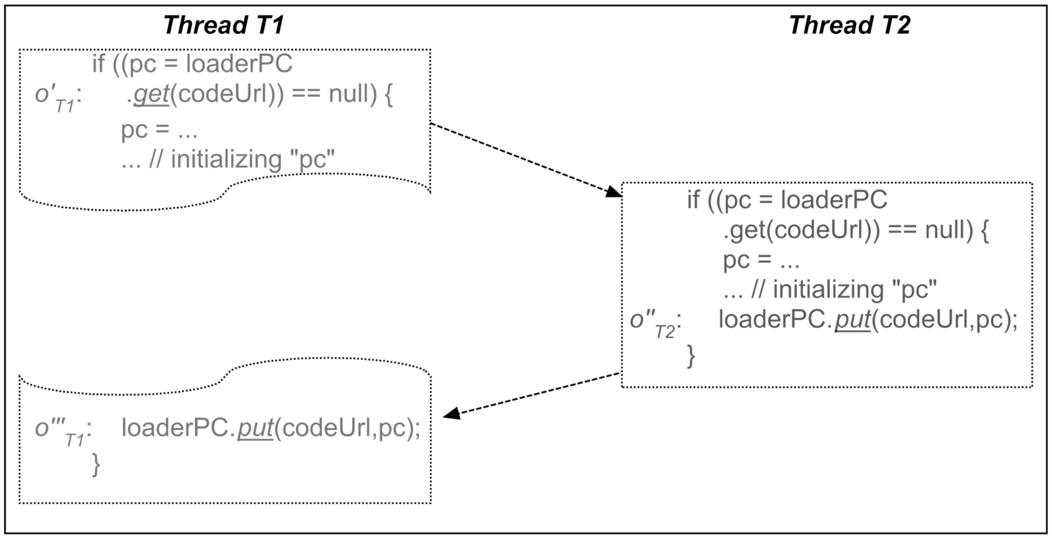
To simplify concurrent programming, the `j.u.c.` package introduces several *thread-safe* concurrent collections, e.g., `ConcurrentHashMap`, `BlockingQueue`, and `CopyOnWriteArrayList`. The interleaving scenario illustrated above for `HashMap.put` cannot happen for `ConcurrentHashMap`, since `ConcurrentHashMap`'s methods are thread-safe and execute atomically.

Before the introduction of `j.u.c.`, a programmer could create a thread-safe `HashMap` using a synchronized wrapper (e.g., `Collections.synchronizedMap(aMap)`). The synchronized `HashMap` achieves its thread-safety by protecting all accesses to the map with a common lock. This results in poor scalability when multiple threads try to access different parts of the map simultaneously, since they contend for the same lock. Lets revisit the example introduced in Fig. 1.1(a). Suppose that a programmer surrounded the whole CHECK-THEN-ACT idiom with a synchronization block which relies on a common lock to protect accesses to the whole map. While this solution ensures atomicity, it prevents any other concurrent accesses to the map, even to the entries that are not currently updated by `put`. This limits the amount of concurrency in the application, which can hurt the performance. We call this a performance bug.

The concurrent collections include the API methods offered by their corresponding non-thread safe counterparts. In addition, they contain new APIs that encapsulate compound update operations, and execute atomically, without resorting to one common lock. Using the concurrent collections over the synchronized collections offers dramatic scalability improvements [10]. However, it is still possible to introduce bugs when using concurrent collections.

Compound CHECK-THEN-ACT operations can lead to atomicity violation bugs. Atomicity (also referred to as serializability [32]) is a property where several concurrently executed operations have an effect that is equivalent to that of a serial execution. As pointed out by Flanagan et al. [33], atomicity is a non-interference property stronger than freedom from data races. (atomicity is a non-interference property that if a code block is atomic, any interaction between that block and other threads should not change the program's behavior). Atomicity violations occur when programmers make wrong assumptions about the atomicity scope of a code block, incorrectly splitting it in two or more atomic blocks and allowing them to be interleaved with other atomic blocks. Atomicity violation of a compound CHECK-THEN-ACT operations occurs in a sequence of three concurrent operations o'_{T_1} ... o''_{T_2} ... o'''_{T_1} such that:
1. o'_{T_1} and o'''_{T_1} are atomic operations executed by one thread (T_1), and o''_{T_2} executes in another thread (T_2) concurrently;
2. o'_{T_1} and o''_{T_2} are data dependent;
3. o'''_{T_1} and o''_{T_2} are data dependent.

To illustrate an atomicity violation, please refer back to the motivating example from Fig. 1.1(a). Fig 2.1 graphically shows a thread interleaving for the code in Fig. 1.1(a). Suppose thread T_1 finds that the map does not contain the key (operation o'_{T_1}), so it will calculate the value and try to put it into the map (operation o'''_{T_1}). Before T_1 puts, it is suspended and another thread T_2 puts a different value to the same key (operation o''_{T_2}). Then T_1 resumes and executes its `put` operation. Under this scenario, the $\langle codeUrl, pc\rangle$ pair put by thread T_2 will be overwritten by the `put` operation o'''_{T_1}. This violates the



```
Thread T1:
        if ((pc = loaderPC
o'T1:       .get(codeUrl)) == null) {
          pc = ...
          ... // initializing "pc"

Thread T2:
        if ((pc = loaderPC
            .get(codeUrl)) == null) {
          pc = ...
          ... // initializing "pc"
o''T2:    loaderPC.put(codeUrl,pc);
        }

Thread T1:
o'''T1:   loaderPC.put(codeUrl,pc);
        }
```

*The arrows represent the order of program execution.

Figure 2.1: An example of buggy interleaving for Fig. 1.1(a).

`put-if-absent` semantics of the original code. In this example, a block which is intended to be atomic in thread T_1 to express `put-if-absent` semantics is split into two blocks by thread interleaving. Thus, there is an atomicity violation.

Notice that in all the CHECK-THEN-ACT atomicity violations that we present, the bug can manifest with as few as two threads. Other papers [34] also show that 96% of concurrency bugs can manifest with just two threads.

Terminology: In this paper we use the term *idiom* to refer to a recurring programming construct that developers use when working with concurrent collections. Like design patterns [35], the idioms abstract away the details from code. We call an *instance of an idiom* a concrete incarnation of the idiom in real code.

The widely-used CHECK-THEN-ACT idiom can be expressed as specific idioms for specific collections (e.g., `put-if-absent` for `ConcurrentHashMap`). An idiom, can also have syntactical variations (e.g., by using different API methods), even for the same collection.

We classify an idiom as *misused* when it can result in a non-atomic execution of the check and act operations (semantic problems) or it is over-synchronized (performance problems). In some cases, this can manifest as a disuse of the atomic library APIs or an erroneous use, in others as over-zealous synchronization. We simply call all of them a misuse of the

Table 2.1: Overview of CHECK-THEN-ACT idioms.

Check / Act upon	Reference	Object state
Reference	(C1) lazy-initialization	(C2) re-initialization
The state pointed by the checked object	(C3) non-null check	(C4) put-if-absent, get, remove, replace, add-if-absent
State other than the one checked	(C5) multi-variable	(C6) remove-if-not-empty

*The columns show what is checked, and the rows show what is acted upon.

concurrent collection API.

2.2.2 Overview of idioms

Based on the definitions we introduced in the previous subsection, we summarize the properties of the CHECK-THEN-ACT idioms that can lead to atomicity violations when using concurrent collections.

The `check` operation could query (i) the reference pointing to the collection (e.g., whether the reference is `null`), or (ii) the state of the collection (e.g., whether a `map` contains a given `key`).

The `act` operation could access (i) the reference pointing to the collection (e.g., make the reference point to a new collection object), (ii) the state of the collection w.r.t. the object queried in the `check` (e.g., put a new $\langle key, value \rangle$ in the `map` on the `key` entry previously checked), or (iii) the state of the collection disregarding any object queried in the `check` (e.g., remove all entries from a `map`).

Thus, using the above classification, there are 6 combinations of `check` and `act` operations, shown in Tab. 2.1. Next we describe each cell giving names to the idioms we have encountered.

Cell C1 refers to `lazy-initialization` idiom, in which the check operation determines whether the reference pointing to the collection is `null`, and if so it creates and assigns a new collection object to the collection reference. Cell C2 refers to an idiom where based on the size of the collection, the act operation can reallocate a new collection. Though the idiom of C2 is rarely discussed in the literature, it still exists in practice and we think it is important.

Cell C3 contains a `non-null check` idiom, e.g., if the collection reference is not `null`, invoke a method on it. Cell C4 contains several idioms. These idioms check if the collection contains a certain entry, and based on the result, they can add/get/remove/override/ the value a new entry.

Notice that Cell C5 refers to invariants that must be maintained across multiple variables [31]. For example, the check determines whether a collection is not `null` and the act updates another collection with the elements of the first collection. Cell C6 contains an idiom where the check determines whether the collection is not empty and the act removes elements.

The structure of Tab. 2.1 is complete: it covers all possible scenarios that can occur in CHECK-THEN-ACT idioms. However, the kinds of idioms that are present in each cell is not complete. We expect this list of idioms could grow if future research explores more case study programs. We mined the idioms that we present in Tab. 2.1 using the 28 open-source applications in our code corpus.

Notice that in our code corpus we did not find examples of idioms in cells C2, C3, and C5, although there are examples in the literature (e.g., C3 in [34], C5 in [31]).

Developers or researchers could use our Tab. 2.1 to manually look for CHECK-THEN-ACT atomicity violations in their code or to design bug detection tools. Although we have observed the check and act operations on instances of collections, similar operations can appear on arbitrary objects that are accessed concurrently.

In the next subsections we provide several examples of usage and misusage of the idioms in Tab. 2.1. Notice that we are intentionally focusing only on concurrent collections in this paper, since the use of concurrent collections implies that the code will be executed by multiple threads. We do not analyze sequential data-structures such as `HashMap` because it is much harder to reason about whether the accesses to these sequential data-structures may happen in parallel with other code.

```
(a)Value v = map.get(key);
   if(v == null){
      v = calc();
      map.put(key, v);
      ...
   }

(b)if(map.get(key) == null){
      v = calc();
      map.put(key, v);
      ...
   }

(c)Value v = map.get(key);
   if(v != null){
      ...
      return;
   }
   v = calc();
   map.put(key, v);
   ...
```

```
(d)if(map.get(key) != null){
      ...
      return;
   }
   v = calc();
   map.put(key, v);
   ...

(e)if(!map.containsKey(key)){
      v = calc();
      map.put(key, v);
      ...
   }

(f)if(map.containsKey(key)){
      ...
      return;
   }
   v = calc();
   map.put(key, v);
   ...
```

Figure 2.2: `Put-if-absent` idiom and its variations for `ConcurrentHashMap`.

2.2.3 Misused Check-then-Act in ConcurrentHashMap

`ConcurrentHashMap` is a thread-safe implementation of `HashMap`. In addition, it contains three new APIs: `putIfAbsent(key, value)`, `replace(key, oldValue, newValue)`, and a conditional `remove(key, value)`. For example, `putIfAbsent` (1) checks whether the map contains a given *key*, and (2) if absent, inserts the $\langle key, value \rangle$ entry. This is a classic example of a CHECK-THEN-ACT idiom. The library guarantees that these two steps are done atomically.

Next, we present the misused CHECK-THEN-ACT idioms when using `ConcurrentHashMap`. Figure 2.2 presents examples of code where the programmer meant to use `put-if-absent` semantics. Notice there are many variations. Figure 2.2(a) shows a temporary variable that is used to hold the result of the check. The check statement can use either `get` (Fig. 2.2(b)) or `containsKey` (Fig. 2.2(c)). Figure 2.2(e) and 2.2(f) show variations where the check condition is reversed. Notice that all of these variations have a `put` invocation that is control-dependent on a `get` or `containsKey` invocation on the same map.

Figure 2.3 and 2.4 show other misused CHECK-THEN-ACT idioms. Unlike Fig. 2.2 that

```
Value v = m.get(key);
if(v == null){
   v = calc();
   map.putIfAbsent(key, v);
}
return v;
```

(a) Put-if-absent idiom with the use of *putIfAbsent* API

```
if(map.containsKey(key)){
   Value v = map.get(key);
   v.m();
   ...
}
```

(b) Get idiom of ConcurrentHashMap

```
(1)Value v = map.get(key);
   if(v != null){
      v = map.remove(key);
      v.m();
      ...
   }

(2)Value v = map.get(key);
   if(v != null &&
      v.equals(v2)){
      map.remove(key);
      ...
   }
```

(c) Remove and conditional-remove idiom

```
(1)Value v = map.get(key);
   if(v != null){
      v = calc();
      map.put(key, v);
      ...
   }

(2)Value v = map.get(key);
   if(v != null && v.equals(v2)){
      v = calc();
      map.put(key, v);
      ...
   }
```

(d) Replace and conditional-replace idiom

Figure 2.3: Other CHECK-THEN-ACT idioms for ConcurrentHashMap.

shows many syntactic variations of the same idiom, the subsequent figures only show one variation for each idiom.

Figure 2.3(a) shows that even when programmers use the new `putIfAbsent` operation instead of the old `put`, they still make mistakes. Notice that the code later uses the value that the programmer assumed to be mapped with the key. Now we describe an interleaving that results in an atomicity violation. After T_1 found that the map does not contain the `key`, it calculates the value `v` and stores it to a reference that is later used. Before T_1 executes the `putIfAbsent` operation, thread T_2 puts another value to the same `key`. Then T_1 resumes, and its invocation of `putIfAbsent` will fail (since the `key` has been mapped by T_2). The last statement returns the reference to the stale value, which is not in the map.

Figure 2.3(b) shows an idiom involving the `get` operation. The code first checks that the map contains a given key, and then invokes a method on the value mapped to this key. An atomicity violation will occur when thread T_1 finds that the map contains the given key. Then T_2 removes the key, and subsequently, T_1 dereferences a `null` value. The code will throw a `NullPointerException`.

Figure 2.3(c) shows the idioms that remove elements. The first idiom (Fig. 2.3(c-1)) removes a $\langle key, value \rangle$ pair if the map contains the key, then subsequent statements use the removed value. Suppose thread T_1 finds that the map contains the key. Before it removes this $\langle key, value \rangle$, it suspends and T_2 removes the same pair. When T_1 resumes, its `remove` invocation returns a `null` value. Thus the subsequent statement that uses the value will throw a `NullPointerException`.

The second idiom (Fig. 2.3(c-2)) is a typical *conditional* removal. The code removes a $\langle key, value \rangle$ pair only if the key is mapped to a specific value `v2`. The atomicity violation occurs if T_2 puts another value (say `v3`) to the same key, after T_1 passed the check, but before it removed the pair. When T_1 resumes, the condition `v.equals(v2)` no longer holds, yet T_1 still removes the pair.

Figure 2.3(d) shows idioms that replace existing elements. These can be seen as complementary to `put-if-absent` semantics, since they have a `put-if-present` semantics. The atomicity violations will occur when thread T_2 removes the $\langle key, value \rangle$ pair while T_1 passed the check, and is about to perform the `put`. The second idiom (Fig. 2.3(d-2)) is a typical *conditional* replace operation.

2.2.4 Misused Check-then-Act in Queues

The `j.u.c.` package contains several thread-safe implementations for working with queues. `ConcurrentLinkedQueue` is a traditional FIFO queue. Its queue operations do not block: if the queue is empty, the retrieval operation returns `null`. The package also provides `BlockingQueues` to add blocking semantics to retrieval and insertion operations. If a queue is empty, the retrieval operation will block until an element is available.

Figure 2.4(a) shows the `remove-if-not-empty` semantics. The code first checks whether the queue contains some elements, and then it removes elements and uses them for further actions. Notice that there are several variations: the check statement can be an `if` or `while` statement, the check operation can query the size of the queue (e.g., `q.size() != 0` or `!q.isEmpty`) or `peek` inside to find elements. The act statement could use `poll, remove, take`, etc.

```
while(!queue.isEmpty()){
   // or while(queue.size()
   // > 0)
   Element e = queue.poll();
   // or remove(), take()
   e.m();
   ...
}
```

(a) Remove-if-not-empty idioms for concurrent queues

```
(1)if(!list.contains(e)){
      list.add(e);
   }

(2)while(!list.isEmpty()){
   // or if(!list.isEmpty())
      Element e = list.remove(0);
      // or list.get(0);
      e.m();
   }
```

(b) Add-if-absent and remove-if-not-empty idioms for CopyOnWriteList

```
if(collection == null){
    collection =
       createCollection();
    collection.add(element);
    ...
}
```

(c) Lazy-initialization idiom for concurrent collections

```
synchronized(map){
   Value v = map.get(key);
   if(v == null) {
      v = calc();
      map.put(key, v);
      ...
   }
}
```

(d) Over-synchronization idiom

Figure 2.4: CHECK-THEN-ACT idioms of other types.

Here we describe one scenario for atomicity violation. Suppose the queue contains only one element and both threads T_1 and T_2 check the condition and find it is not empty. The thread that is the last to invoke the retrieval operation will get a `null` value which makes the code throw a `NullPointerException`.

2.2.5 Misused Check-then-Act in Lists

The `j.u.c.` package contains a thread-safe implementation for working with lists. `CopyOnWriteArrayList` is a data structure in which all mutative operations (e.g., `add`) are implemented by making a fresh copy of the underlying array. Iterators iterate over a *snapshot* view of the collection at the point that the iterator was created.

Figure 2.4(b) shows two idioms. The first idiom (Fig. 2.4(b-1)) illustrates `add-if-absent` semantics. The code appends an element to a list, if the list does not already contain it. Two threads, T_1 and T_2 can both pass the check at the same time, and they will append the same element twice.

The second idiom (Fig. 2.4(b-2)) illustrates the `remove-if-not-empty` idiom, and the atomicity violation happens under the same interleaving as shown in Sec. 2.2.4

2.2.6 Misused Check-then-Act in Lazy Initialization

The `lazy-initialization` idiom is also error-prone. Figure 2.4(c) shows code that lazily creates a concurrent collection when it is needed. However, code also adds some elements into it. The atomicity violation will occur if both T_1 and T_2 find the collection reference is `null` and initialize it. In this case, one initialization will override the other. Now the elements added by T_1 are no longer seen by T_2.

2.2.7 Over-Synchronization in Check-then-Act

Figure 2.4(d) shows a `put-if-absent` idiom wrapped by a synchronization block. Assuming that the other accesses to the map are protected by the same lock, this code is properly synchronized, thus the idiom executes atomically. However, the synchronization degrades the performance: it prevents threads who are working on different buckets of the map to operate in parallel. This defies the entire purpose of using a concurrent collection.

2.2.8 Correction of Idioms

A developer can use two ways to correct the atomicity violations caused by misused CHECK-THEN-ACT idioms: (1) leveraging the proper atomic API provided by the concurrent collections, or (2) adding a synchronization block around the CHECK-THEN-ACT code.

Figure 2.5 shows the strategies that we use to fix the misused CHECK-THEN-ACT idioms. We underlined the statements that we add or change. For the idioms that have `put-if-absent` semantics, we use the `putIfAbsent` operation instead of `put`. When the code further reads the value placed in the map, our fix ((Fig. 2.5(a)) checks the status of the `putIfAbsent` to judge whether the assumed value was indeed placed in the map (`putIfAbsent` returns `null` to indicate successful execution). Note that for `put-if-absent` idiom with the use of `putIfAbsent` method, our fix also checks the status of the `putIfAbsent`.

```
Value v = map.get(key);
if(v == null) {
   v = calc();
   Value tmpV = map.putIfAbsent(key, v);
   if(tmpV != null)
      v = tmpV;
}
... // variable  v is used here
```

(a) Fix for put-if-absent idiom

```
Value v = map.get(key);
if(v != null) {
   v = map.remove(key);
   if(v != null) {
      v.m();
      ...
   }
}
```

(b) Fix for remove idiom

```
Value v = map.get(key);
if(v != null) {
   v = calc();
   Value tmpV = map.
      replace(key, v);
   if(tmpV != null) {
      ...
   }
}
```

(c) Fix for replace idiom

```
Value v = map.get(key);
if(v != null) {
   v.m();
   ...
}
```

(d) Fix for get idiom

```
while(!queue.isEmpty()) {
   Element e = queue.poll();
   if(e != null) {
      e.m();
      ...
   }
}
```

(e) Fix for idiom of queue

Figure 2.5: Fixes for CHECK-THEN-ACT idioms.

In the fixes in Fig. 2.5(b), 2.5(c) and 2.5(e), CTADETECTOR adds code to check the return value of the `act` operation, thus preventing `NullPointerExceptions`. For the `get` idiom in Fig. 2.3(b), we replace the use of `containsKey` with checking whether the mapped value is not `null`.

Note that we do not show the fixes for the `add-if-absent` and `lazy-initialization` idioms. The fix for the former is similar to `put-if-absent`, while the fix for the latter is wrapping the idiom with a proper synchronization block. Although adding synchronization blocks is beyond the scope of CTADETECTOR, techniques such as atomic region identification [36] or automated atomicity violation fixes via static analysis and testing [37] can be employed to help add such synchronization blocks. To fix the performance bugs because of over-synchronization, CTADETECTOR removes the lock and uses the corresponding compound update API method. For example, in Fig. 2.4(d), CTADetector removes the synchronization and uses `putIfAbsent` instead of `put`.

Notice that the fixes we propose are not the only way to fix the buggy idioms. Consider the example in Fig. 2.5(a). For the correctness sake, the first check (`if (v == null)`) is redundant, since the `putIfAbsent` will perform its own `null` check, so the check-then-act idiom is guaranteed to be atomic. However, by keeping the original check in place, we reduce the chance of wasting memory and computation by not creating the unnecessary value `v` especially in cases when the map already contains this entry. Another alternative is to remove the first check from the code, but this has the effect of always creating the object `v`, regardless of whether the map already contains this entry. Other solutions, even more involved exist, for example to use a wrapper around the value class that performs lazy initialization (thus delaying the execution of `calc()`) until the first access. One way to do this is to declare the map as `Map<Key, Future<Value>>` [38]. However, we feel that our solution is the least intrusive to the current code.

2.3 Analysis of Idiom Instances

In this section we answer four research questions:

- **RQ1:** What are the commonly used check-then-act idioms in real-world programs?
- **RQ2:** Which idioms are the most error-prone?
- **RQ3:** Do misused idioms result in real bugs? Are our patches accepted by developers?
- **RQ4:** What is the evolution of programs w.r.t. check-then-act idioms?

RQ1 and RQ2 help us, library designers, and tool builders learn about the state of the practice. RQ3 evaluates whether the found misused idioms are critical for the correctness or performance of real world programs. RQ4 shows whether developers pay more attention to check-then-act idioms.

2.3.1 Experimental setup

Subjects: To answer the first three research questions, we used a corpus of 28 real-world open-source programs. The first three columns of Table 2.2 show the subject programs, the version number, the size – in non-blank, non-comment source lines of code (SLOC)[1], and the domain of application. All programs use concurrent collections. For each program, we use the most current version at the time of the experiments.

To study the evolution of the programs (RQ4), out of the initial corpus, we selected those projects that had multiple releases between 2007 and 2012. This created a corpus of 18 programs. For each program, we chose three major releases: V_3 – the most current release (as shown in Tab. 2.3), V_2 – a major release from 2010–2011, and V_1 – a major release from 2007–2009.

Process: We ran our tool, CTADetector, over our corpus. CTADetector classified idioms as correct or misused. The latter contains semantic or performance issues. We manually verified the results and sorted them based on the idioms that we introduced in Sec. 2.2.

To confirm whether the misused idioms result in real bugs, we reported 155 instances to the open-source developers. Our companion website [40] contains links to our bug reports. Along with the bug description, we also submitted a patch generated by CTADetector. When developers reported that a misused idiom does not result in a real bug, we further asked them to elaborate why the atomicity violation in the idiom is acceptable for their program.

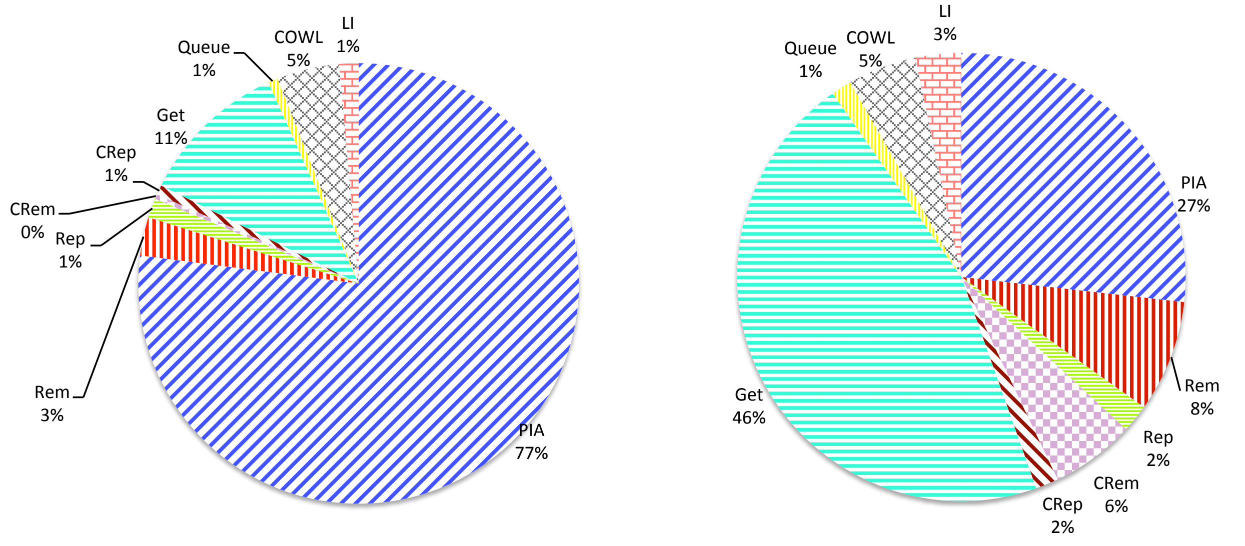
To answer the evolution question we compare the number of correct and misused instances of idioms along the three major releases.

2.3.2 Results

RQ1: What are the commonly used check-then-act idioms in real-world programs?

Fig. 2.6 shows the distribution of correct and misused idioms across the corpus of 28

[1]as reported by the SourceCounter [39] tool



(a) The distribution of misused idioms. The total number is 283.

(b) The distribution of correct idioms. The total number is 551.

*PIA: `put-if-absent`, Rem: `remove`, Rep: `replace`, CRem: `conditional-remove`, CRep: `conditional-replace`, Get: `get`, Queue: idioms for queues, COWL: .idioms for lists, LI: `lazy-initialization`.

Figure 2.6: The distribution of idioms.

projects. CTADetector found 282 instances of misused idioms and 545 instances of correct idioms.

Notice that in each category, there is one idiom that clearly dominates the others: the `put-if-absent` idiom is the most common misused idiom, while `get` is the most common correctly used idiom. It is also surprising that the top four idioms in each category are different.

While `j.u.c` provides 14 different kinds of concurrent collections including 2 maps, 2 sets, 9 queues, and 1 list, our result shows that 93% (264) of the misused instances and 90% (492) of the correct instances appear when using `ConcurrrentHashMap`. This is expected: (i) a previous study [11] shows that `ConcurrrentHashMap` is the most widely used concurrent collection in Java, and (ii) `ConcurrrentHashMap` stores $\langle key, value\rangle$ pairs so it offers a richer API than other collections, thus there are more choices to compose operations.

Table 2.2: Correctly used CHECK-THEN-ACT idiom instances in 28 real-world programs.

Subject	SLOC	Domain	PIA	Rem	Rep	CRem	CRep	Get	Queue	COWL	LI	**Total**
Annsor 1.0.3	1605	Annotation processor	-	-	-	-	-	-	-	-	-	0
Cassandra 1.1.1	132183	Distributed database	5	1	-	-	-	8	1	-	-	15
CXF 2.6.1	441269	Services framework	9	1	-	1	-	8	1	6	2	28
Lucene 4.0.0	361494	Text search engine	1	-	-	-	-	4	-	-	-	5
Mina 2.0.4	46435	Network framework	2	-	-	-	-	5	1	-	-	8
Struts 2.3.4	146919	Web framework	2	-	1	3	1	1	-	-	-	8
Tomcat 7.0.28	215298	Servlet container	5	-	-	-	-	10	2	-	-	17
Trinidad 2.0.1	220484	JSF framework	5	2	-	1	-	8	-	-	4	20
Wicket 1.5.7	169142	Web framework	3	-	-	-	-	7	-	-	-	10
BlazeDS 4.0.1	68887	Web messaging	2	3	-	-	-	11	-	16	9	41
Carbonado 1.2.3	54254	Persistence layer	2	-	-	2	-	2	-	-	-	6
CBB 1.0	17001	Building Blocks	-	-	-	-	-	-	-	-	-	0
DWR 1.1	35630	Ajax for Java	10	1	-	-	1	3	-	-	-	15
Ektorp 1.1.1	10112	CouchDB API	-	-	-	-	-	-	-	-	-	0
Flexive 3.1.6	139011	Content management	8	-	4	-	1	2	-	-	-	15
Glassfish 3.1	721944	Application server	11	7	-	1	-	33	1	-	1	54
Granite 2.3.2	41790	Data Service	14	1	-	-	-	10	-	-	-	25
Hazelcast 2.0.4	89080	Data distribution	13	4	-	3	-	39	-	-	-	59
Ifw2 1.33	55596	Web framework	5	-	-	-	-	1	-	-	-	6
JBoss AOP 2.2.2	196106	AO framework	11	-	-	-	-	10	-	-	-	21
JSefa 0.9.3	18173	Object serialization	3	-	-	-	-	2	-	-	-	5
Memcache	6695	Caching system	-	1	-	-	1	8	-	-	-	10
Open EJB 4.0.0	286451	EJB container	2	4	-	2	-	5	-	-	-	13
Open JDK	2262000	JDK 8	24	8	6	16	5	19	-	2	-	80
RestEasy 2.3.4	123813	JAX-RS client	4	-	-	2	-	9	-	-	-	15
Tersus	113260	Visual programming	1	-	-	-	-	-	-	-	-	1
Vo Urp	29954	Data models translator	1	-	-	-	-	-	-	-	-	1
Zimbra	448573	Collaboration server	3	11	-	2	-	44	2	3	2	67
Total	**6453159**		**146**	**44**	**11**	**33**	**9**	**249**	**8**	**27**	**18**	**545**

*Columns 4 to 12 represent `put-if-absent`, `remove`, `replace`, `conditional-remove`, `conditional-replace`, `get` idioms, idioms for queues, `CopyOnWriteList` and `lazy-initialization`.

Table 2.3: Misused CHECK-THEN-ACT idiom instances in 28 real-world programs.

Subject	PIA		Rem		Rep		CRem		CRep		Get		Queue		COWL		LI		**Total**	
	s	p	s	p	s	p	s	p	s	p	s	p	s	p	s	p	s	p	s	p
Annsor 1.0.3	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0
Cassandra 1.1.1	3	-	-	-	-	-	-	-	-	-	2	-	1	-	-	-	-	-	6	0
CXF 2.6.1	5	4	-	-	-	-	-	-	-	-	3	-	-	-	2	-	1	-	11	4
Lucene 4.0.0	1	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	2
Mina 2.0.4	2	2	-	-	1	-	1	-	1	1	-	-	-	-	-	-	-	-	5	3
Struts 2.3.4	5	1	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	6	1
Tomcat 7.0.28	2	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-	1	-	5	0
Trinidad 2.0.1	14	2	-	-	-	1	-	-	-	-	1	-	-	-	-	-	-	-	15	3
Wicket 1.5.7	10	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	11	0
BlazeDS 4.0.1	1	3	2	-	-	-	-	-	-	-	6	-	-	-	1	-	-	-	10	3
Carbonado 1.2.3	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2	0
CBB 1.0	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4	0
DWR 1.1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0
Ektorp 1.1.1	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4	0
Flexive 3.1.6	10	2	-	-	-	-	-	-	-	-	2	-	-	-	-	1	-	-	12	3
Glassfish 3.1	5	6	2	-	-	-	-	-	-	-	2	-	-	-	1	-	1	-	11	6
Granite 2.3.2	5	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	5	2
Hazelcast 2.0.4	14	3	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	-	16	3
Ifw2 1.33	2	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	-	4	0
JBoss AOP 2.2.2	11	6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	11	6
JSefa 0.9.3	1	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	2
Memcache	6	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	-	8	0
Open EJB 4.0.0	6	5	2	-	-	-	-	-	-	-	4	1	-	-	-	-	-	-	12	6
Open JDK 8	19	3	-	-	-	-	-	-	-	-	-	-	1	-	1	-	1	-	22	3
RestEasy 2.3.4	11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	11	0
Tersus	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0
Vo Urp	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3	0
Zimbra	14	12	2	-	-	2	-	-	-	-	-	2	-	-	3	2	-	-	19	18
Total	**163**	**55**	**8**	**0**	**1**	**3**	**1**	**0**	**1**	**1**	**28**	**3**	**2**	**0**	**10**	**3**	**4**	**0**	**218**	**65**

*For each idiom, the columns `s` and `p` show the number semantic and performance bugs, respectively. Columns 2 to 10 represent `put-if-absent`, `remove`, `replace`, `conditional-remove`, `conditional-replace`, `get` idioms, idioms for queues, `CopyOnWriteList` and `lazy-initialization`.

RQ2: Which idioms are the most error-prone?

Table 2.2 shows the number of correctly used instances of idioms for each project. Columns 2–10 in Tab. 2.3 show the number of misused instances of idioms for each project, while column 11 shows the total number for each project. For columns 2–11 in Tab. 2.3, sub-column `s` shows the number of semantic bugs while sub-column `p` shows the number of performance bugs. By comparing Tab. 2.2 and 2.3, we can notice that with the exception of the `put-if-absent` idiom, the number of correct instances outweighs the misused instances to a large extent for most projects. This means most developers are aware of how to correctly use the concurrent collections, but may make mistakes occasionally.

For `put-if-absent` idiom, the number of misused instances is larger than the number of correct instances. This shows that this idiom is the most error-prone. However, as we show in RQ3, not all misused instances are perceived as buggy by the developers.

For the misused idioms, Tab. 2.3 shows 65 instances of idioms (column `Total`, sub-column `p` in Tab. 2.3) where the developers wrapped the CHECK-THEN-ACT within a synchronized block. However, this is over-synchronization, and we classify these instances as performance bugs. This shows that some developers know there are atomicity violations in the idioms, but they add synchronization to avoid them, instead of using the atomic APIs from the concurrent collections. In contrast, CTADetector correctly suggests patches that involve the atomic APIs, as discussed in Sec 2.2.8. This can dramatically improve the performance.

RQ3: Do misused idioms result in real bugs? Are our patches accepted by developers?

In our corpus of projects, we selected the 17 most active projects. We reported the misused idioms and also provided the patches generated by CTADetector. For some large projects like GlassFish, we did not report all the misused idioms that are detected by CTADetector, but only those for the major components.

For the 17 projects that we contacted, we reported 155 bugs. However, we only got replies from the developers of 11 projects. We reported the bugs between June 2012 and August 2012, and we received no further responses after March 2013. Table 2.4 shows these 11 projects, along with the number of reported and replied bugs in each project (column 2), and the number of bugs confirmed by developers (column 3). Out of the 90 bugs that were investigated by developers, they confirmed 49 semantic and 11 performance bugs, so clearly these are real problems. The developers of 9 projects accepted our patches for these 60 bugs and included the patches in the new versions. Last column shows the version numbers that include our patches.

As shown in Table 2.4, not all of the misused idioms lead to bugs, although two thirds of the instances cause buggy behaviors or slow response in the programs. For the remaining one third of our reported misused idioms, the developers do not think these cause problems. We provide such examples in Fig. 2.7, in which we label the *check* and *act* operations.

For the case of semantic bugs that developers did not confirm, we divide the reasons that the developers provided into three categories:

```
        public void maybeInitializeLocalState(int generationNbr) {
            EndpointState localState = endpointStateMap.
                get(FBUtilities.getBroadcastAddress());

chk:       if ( localState == null ) {
               HeartBeatState hbState = new
                   HeartBeatState(generationNbr);
               localState = new EndpointState(hbState);
               localState.markAlive();
act:           endpointStateMap.put(
                   FBUtilities.getBroadcastAddress(), localState);
           }
        }
```

(a) *Gossiper.java* in *Apache Cassandra*

```
        public void manageApp(Context context)  {
            String contextName = context.getName();

chk:       if (deployed.containsKey(contextName))
                return;

            DeployedApplication deployedApp = new
                DeployedApplication(contextName);
            ...

act:    deployed.put(contextName, deployedApp);
        }
```

(b) *HostConfig.java* in *Apache Tomcat*

```
        private static Class<?> getSpiClass(String type) {
            Class<?> clazz = spiMap.get(type);
chk:        if (clazz != null) {
                return clazz;
            }
            try {
                clazz = Class.forName("java.security." + type + "Spi");
act:            spiMap.put(type, clazz);
                return clazz;
            } catch (ClassNotFoundException e) {
                throw new AssertionError("Spi class not found", e);
            }
        }
```

(c) *Security.java* in *Open JDK 8*

```
        private static boolean isCCLOverridden(Class<?> cl) {
            if (cl == Thread.class)
                return false;
            ...
            WeakClassKey key = new WeakClassKey(cl,
            Caches.subclassAuditsQueue);
            Boolean result = Caches.subclassAudits.get(key);
chk:        if (result == null) {
                result = Boolean.valueOf(auditSubclass(cl));
act:            Caches.subclassAudits.put(key, result);
            }
            return result.booleanValue();
        }
```

(d) *Thread.java* in *Open JDK 8*

```
        protected Set<String> getStandardAttributes() {
            Class clz = getClass();
            Set<String> standardAttributes = standardAttributesMap.get(clz);
chk:        if (standardAttributes == null) {
                standardAttributes = new HashSet<String>();
act:            standardAttributesMap.put(clz, standardAttributes);
                while (clz != null) {
                    for (Field f : clz.getDeclaredFields()) {
                        ...
                        standardAttributes.add(f.getName());
                    }
                    ...
                }
            }
            return standardAttributes;
        }
```

(e) *UIBean.java* in *Apache Struts 2*

Figure 2.7: Examples of buggy instances not confirmed by developers.

Table 2.4: Bug confirmation from the developers: 49 semantic bugs and 11 performance bugs.

Subject Name	Replied Bugs	Confirmed	Fixed Version
Apache Cassandra	6	5	1.1.2
Apache CXF	15	15	2.7.0
Apache Mina	5	5	2.0.5
Apache Struts	7	2	2.3.5
Apache Tomcat	5	4	7.0.30
Apache Trinidad	1	0	-
Apache Wicket	11	11	1.5.8
Glassfish	8	6	4.0
GraniteDS	7	7	3.0.0 beta1
OpenJDK	14	0	-
RestEasy	11	5	2.3.5
Total	**90**	**60**	

1. Impossible interleaving: The buggy interleaving that we described in Sec. 2.2 does not happen in the application context. This can be due to two reasons. First, the code containing the idiom is never executed concurrently. For example, in Cassandra (shown in Fig. 2.7(a)), the developers claimed that "*`maybeInitializeLocalState` shouldn't be called concurrently, but it doesn't hurt to clean that up too*". This was surprising to us, since this defies the whole reason of using a concurrent collection. However, it could be that only some code snapshots that use an instance of a concurrent collection are executed concurrently, or it could be that developers envision some future evolution where the code will indeed run concurrently. Interestingly, although the developers did not confirm this as a bug in this code snippet, they still accepted and applied our patch.

Second, the conflicting operation never executes concurrently. For example, in `Tomcat` (shown in Fig. 2.7(b)), at any given moment, there is only one thread that puts a value in the map. The developers said "*This issue is definitely not valid since a Host will never permit multiple children with the same name at a time. This change was not included in the fix.*"

2. Unique values: For some `ConcurrentHashMap` usages, the program *uniquely* calculates one single `value` for a given `key`. That is, the `value` is either a singleton object [35], or the program can calculate several value objects for the same key, but they are in the same equivalence class. Thus, for the `put-if-absent` idiom, even if the value written by one thread is overwritten by another thread, since the two values are equivalent, the idiom does not lead to bugs. In `Open JDK 8`, there are 13 cases when the values are uniquely calculated from the keys. We show two such examples in Fig. 2.7(c) and (d). The developers said "*In both of the examples you mentioned, the race seems benign. In the first example, a `Class` may be returned that's not in the map anymore if it's overwritten, but `Class.forName()` with exact same args should return the same instance of `Class`. Likewise in the second example the return value is a primitive `boolean`, so even if two threads race, it shouldn't matter. The race would be an issue if identity equality is required and the creation/construction of the object inside the method does not itself guarantee identity, but I don't see either of these issues in these two examples*".

3. Program resilience: The program does not care whether a value written by one thread is overwritten by another thread. For the example from `Struts` (shown in Fig. 2.7(e)), the developers said "*`standardAttributes` of `UIBean` doesn't need atomicity, since I design it just as a cache. Atomicity is not mandatory here*". Since `ConcurrentHashMap` is used as a cache, even if the value is overwritten and no longer in the map, it can still be used without affecting the behavior. For `lazy-initialization` idiom, there is also a case in `GlasshFish` where even if the values put into the map are lost, those values will be created and put again by other threads.

Discussion: In the above cases, the race conditions in the idioms are benign and can improve the performance (e.g., `put` is faster than `putIfAbsent`). Notice that reasoning about such cases requires *deep understanding* of the domain and the concurrency model of the program. This is usually beyond the capabilities of tools and is better left to human expert judgement. This is exactly the reason why CTADetector is *interactive*, allowing the human expert to judge whether the misused idiom is really a bug.

However, developers should carefully check the semantics of the programs to make sure they use an idiom correctly, since as our result shows, 67% of misused instances lead to real bugs. Furthermore, the developers should document the invariants that ensure correctness. This can prevent future versions running afoul precisely because of these bugs. In the 30 instances that developers did not considered real bugs, they documented only one such invariant.

For performance bugs that developers did not confirm, we divide the reasons that the developers provided into two categories:
1. Rarely executed code: The idioms that are over-synchronized are not executed often. For example, the over-synchronized idioms in `GlasshFish` are only executed when an application is deployed to the `GlasshFish` server (which occurs rarely), not when a user makes runtime accesses to that application (which occurs often). Thus, the performance gains obtained by fixing such idioms are not measurable by their performance benchmarks.
2. Unnecessary objects creation: In the fix of `put-if-absent` idiom that we show in Fig. 2.5(a), there is a chance that two threads enter the `if` statement at line 2 concurrently and create two new objects. Notice that only one of the created objects will be put into the `map`, so the code is still correct because it relies on the atomicity guarantees from `ConcurrentHashMap`. For example, in the `GlassFish` project, creating `FileLoggerHandler` objects multiple times can be expensive and the `GlassFish` developers prefer to use synchronization within the idiom in order to avoid unnecessary object creations: *"For `FileLoggerHandlerFactory`, I prefer to keep the existing logic ... I am concerned that it will create more `FileLoggerHandler` objects than necessary".*

Discussion: In the above cases, removing synchronization on the idiom does not provide noticeable performance improvements or may even degrade performance. However, similar to our discussion on semantic bugs, reasoning about such cases requires domain knowledge. Thus, it requires an interactive mode of execution like the one that we envision for CTADetector.

RQ4: What is the evolution of programs w.r.t. check-then-act idioms?

For the 18 projects that have multiple major releases, Table 2.5 shows the total number of instances of idioms across three major releases. Notice that the number of instances increases for both misused and correctly used idioms. This means that developers are embracing concurrent collections. This is consistent with our recent finding [8] that shows that many

Table 2.5: The evolution of idioms

Version / Instances	V_1	V_2	V_3
misused (m)	112	156	201
correct (c)	152	253	418
m/(m+c)	**0.42**	**0.38**	**0.32**

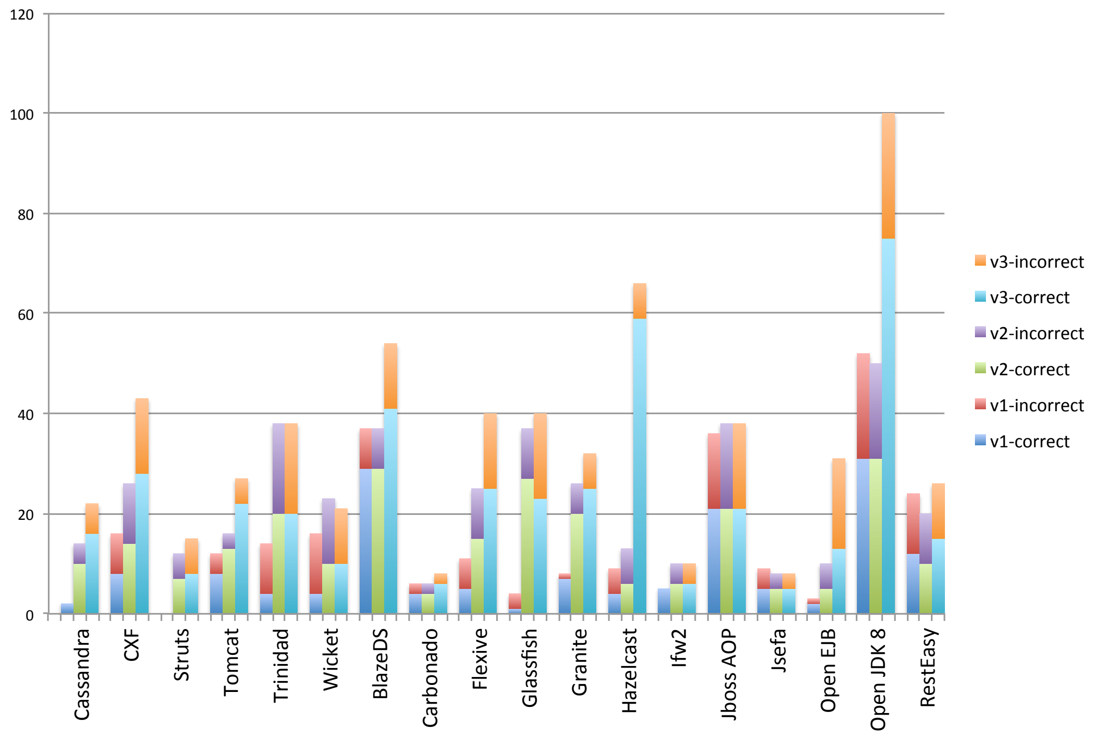
developers are embracing multicore parallel programming.

Interestingly, the ratio of misused instances (as shown by the last row) decreases in later versions. One reason could be that developers pay more attention to the correct usage of CHECK-THEN-ACT idiom, since as time goes by, programmers have more resources to learn how to use the concurrent collection correctly. Another explanation is they found such bugs in production and applied patches.

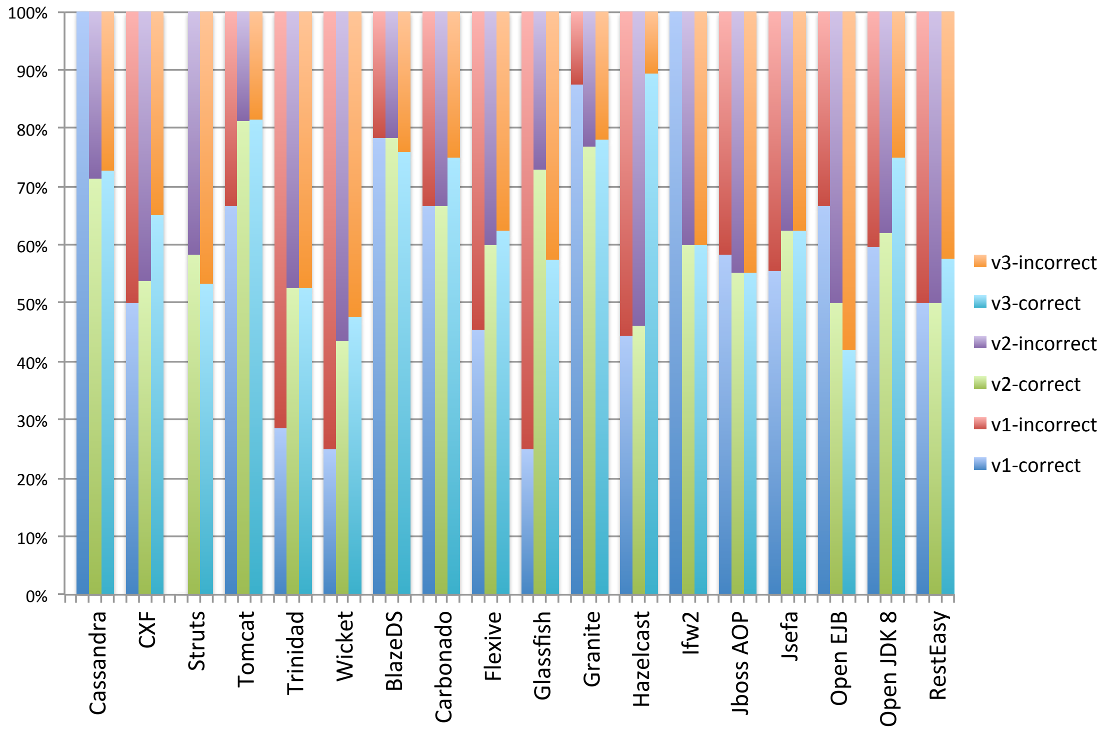
Figure 2.8(a) shows the number and Fig. 2.8(b) shows the ratio of correct and incorrect instances of CHECK-THEN-ACT idioms, for the three versions of each project. These figures show that although for most projects, the ratio of misused instances decreases in later versions, the absolute number of misused instances still increases. On the other hand, there are also some projects, e.g. Open EJB, in which the ratio of misused instances increases slightly in the later versions. This means that CHECK-THEN-ACT usage of concurrent collections is still error-prone. Thus, cataloging the CHECK-THEN-ACT usage of concurrent collections and automating bug finding tools like CTADETECTOR is helpful.

2.3.3 Threats to Validity

Construct Validity: Since we did not run the open-source applications, but instead took a static analysis approach to find misused idioms, one could argue whether such misuses can really result in faults in the application code. What if those code idioms are never executed concurrently? First, we relied on the intuition that the usage of concurrent collections indicates the developers' intent to execute that code concurrently. If there was no intent to invoke those idioms from concurrent threads, the developers should have used the equivalent sequential, non thread-safe collections (e.g., `HashMap` instead of `ConcurrentHashMap`), which have better performance than the thread-safe concurrent collections. Second, for the



(a) Number of correct and incorrect instances



(b) Ratio of correct and incorrect instances

Figure 2.8: The number and ratio of correct and incorrect instances.

majority of reported cases, the developers confirmed the possibility of a thread interleaving that can result in buggy behavior, thus they accepted our patch. Third, even in cases where the developers claim that the bugs cannot occur in practice, we found that the invariants ensuring correctness are not sufficiently documented in the source code to avoid future versions of the software running afoul of precisely these bugs. Thus, it is still valuable to have tools like CTADetector point the location of such potential bugs, so that developers can at least document the code.

With respect to computing the evolution of correct and incorrect idioms shown in Tab. 2.5 and Fig. 2.8, we used the idioms identified automatically by CTADetector. A more precise metric would have been to cross-validate each idiom instance by reporting it to the developers, the same way how we reported 155 bugs for the 17 projects that we used to answer RQ3. Thus, some of the idioms that CTADetector classified as buggy in the first two versions, might not be confirmed by developers. However, reporting all bugs is not feasible. First, developers might not care about confirming bugs in versions of the code that are 6 years old. Second, preparing and submitting thousands of bug reports takes an inordinate amount of time. Third, our aim was to show the trend and the presence of bugs in several versions, and thus the exact number is less important.

Internal Validity: One could ask whether the design of the experiment and the results truly represent a cause-and-effect relationship. For example, how did we mitigate bias during manual inspection of the reported misused idioms and during the bug reporting activities? First, we created a text message in which we described a possible thread interleaving, and we referred to the lines of code where CTADetector identified the misused idioms. It was up to the original developers to examine our report and to confirm whether the buggy interleaving could be possible. In a few cases when their replies were vague, we followed up with the developers with another clarifying question, and only afterward we labeled the idiom as buggy. The developers informed us whether the patch has been applied. In some cases the developers changed the CTADetector-generated patch to confirm to the project coding standards. For example, developers of Cassandra accepted our patch, CASSANDRA-4402; they renamed some local variables, added comments and assertions, and reformatted the code.

When deciding for which projects to submit bug reports, our criteria were (i) the project be active, (ii) have bug repositories, and (iii) have recent activity, e.g., within the last month there was at least a bug reported and discussed.

External Validity: One could ask whether our results are applicable and generalizable to a wider range of software projects, even those not developed in Java. First, we chose a diverse corpus of 28 widely used Java open source projects totaling over 6.4M SLOC. They are developed by very different entities, by large organizations or researchers, and covering domains such as GUIs, servers, databases, search engines, web and desktop, programming tools, etc. We expect that other kinds of applications will still use CHECK-THEN-ACT idioms, and we have no reason to believe that other applications won't have buggy idioms. Second, the other popular object-oriented languages, C++ and C#, have similar libraries containing concurrent collections: TBB [3] library for C++, and Concurrent Collections [2] for C#. These libraries provide very similar APIs. Thus, although the presentation is in the context of Java, the problem is rather common in other languages such as C++ and C#.

Finally, this paper specifically investigates atomicity bugs. However, there are many other kinds of concurrency-related bugs. For example, ordering bugs [34] where programmers implicitly rely on a sequentially consistent memory model (which does not hold in Java), or deadlocks. Although we did not investigate such bugs, we believe that catalogs for such kinds of bugs can serve as excellent educational resources.

Reliability: Is our empirical study reliable and replicable? The corpus and the bug reports, including the discussions with the open-source developers is available on our webpage. Also, CTADETECTOR is open-source and freely available:
`http://refactoring.info/tools/CTADetector/`

2.4 Implications

There are several implications of our study. We organize them based on the community for which they are relevant.

2.4.1 Developers

Becoming proficient with a new library requires a long-term commitment. Developers without concurrent programming experience might ask themselves: should we learn how to correctly use concurrent libraries. Programmers can learn a new programming construct through both positive and negative examples. Our catalog of idioms teaches them how to use CHECK-THEN-ACT idioms correctly. Along with the hundreds of instances of idioms, it provides a tremendous educational resource.

2.4.2 Library Designers

Library designers can use our findings to make the APIs more robust, provide better documentation, or provide new APIs as replacements for those that are error-prone. For example, in the `put-if-absent` idiom shown in Fig. 2.2, if the key is not in the `ConcurrentHashMap`, developers first compute a value for the key and put the $\langle key, value \rangle$ entry pair into the map. To make these compound operations atomic, library designers can provide a new atomic API `compute-if-absent` for `ConcurrentHashMap`. Thus, the `put-if-absent` idiom can be replaced by a single method invocation of `compute-if-absent`. We have been in contact with Doug Lea, the main architect of the `j.u.c.` package, and our finding of so many misuses of the `put-if-absent` idiom has led him to add `compute-if-absent` API in the Java 8 version of `ConcurrentHashMap` (officially released in the first quarter of 2014).

Additionally, library designers should provide better documentation for concurrent collections. In the documentation, newly introduced atomic APIs should be emphasized and examples of using these new APIs should be provided. In this way, programmers are more likely to notice the existence of new APIs.

2.4.3 Researchers

The testing community can focus its efforts to find higher-level atomicity violations bugs in concurrent programs. Although traditionally the testing and verification community have focused mostly on low-level data races, our findings show that high-level atomicity violations are an important class of bugs and are widespread in today's programs.

We hope that our paper inspires the community to identify other kinds of CHECK-THEN-ACT idioms used in concurrent collections. Moreover, alhough our work focuses on the CHECK-THEN-ACT idioms of concurrent collections, CHECK-THEN-ACT bugs can also happen on arbitrary objects. To detect CHECK-THEN-ACT bugs for other kinds of objects, the approach should first determine if an object may be accessed by multiple threads. Then the approach should identify the `check` and `act` operations of the shared object, by analyzing if they access the same field of the shared object.

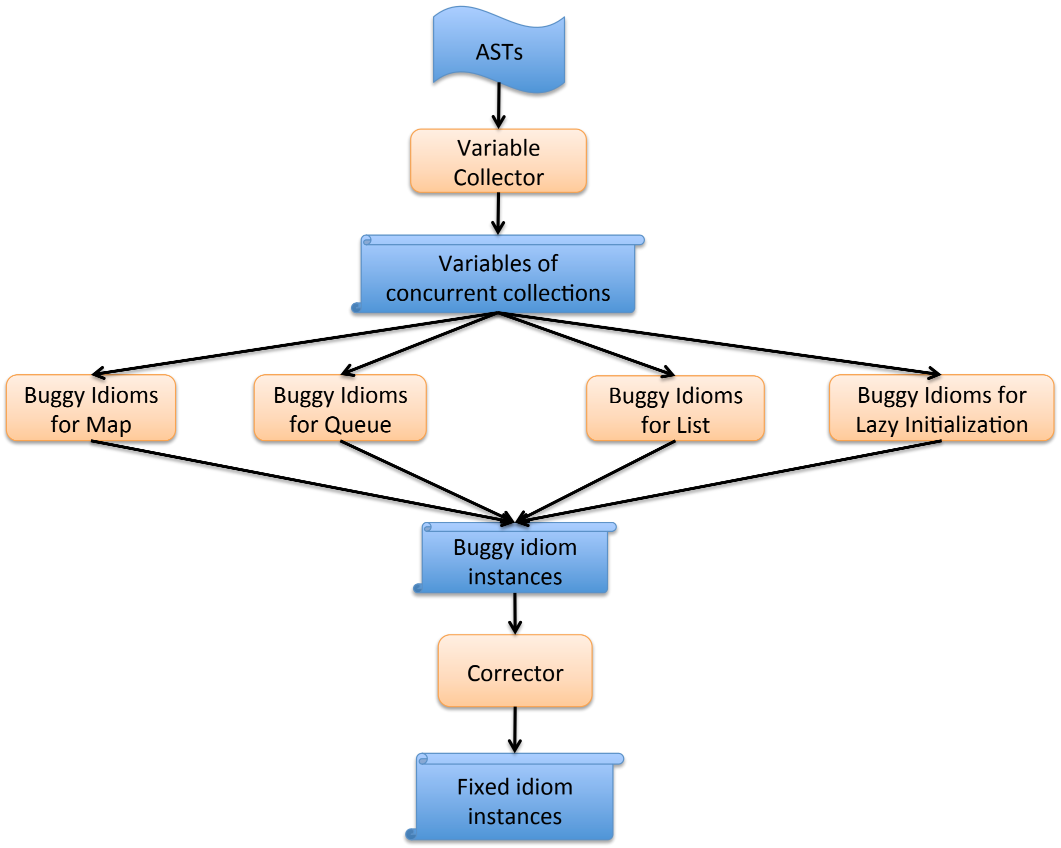
2.4.4 Tool Builders

The answer of RQ3 shows that not all detected CHECK-THEN-ACT idiom instances are considered to be bugs by developers. Since concurrency bug detection tools need to produce few false warnings if they are ever to be used by practitioners, our finding raises several other practical questions for tools builders: (i) What to do when an idiom would be an error if the component was used concurrently, but it is now used sequentially? Should this be reported as a warning? (ii) Do two writes of the same value constitute a race? (iii) Can racy programs be "correct"? These questions are important for nearly any concurrency bug detection tools. Tool builders should investigate the techniques that can mitigate these issues.

2.5 Analysis Infrastructure

In this section we describe our approach to automatically detect and correct the CHECK-THEN-ACT idioms that we listed in Sections 2.2–2.3. Subsection 2.5.1 shows an overview of the architecture and outputs of CTADETECTOR. Subsection 2.5.2 presents the detection and correction approach. Subsection 2.5.3 presents the pseudo-code for detecting one of the idioms (`put-if-absent`). Subsection 2.5.4 discusses the limitations and possible improvements.

2.5.1 Infrastructure Overview



*The rounded rectangles are the components while the others are the inputs/outputs of each component.

Figure 2.9: The architecture of CTADetector.

We implemented both the detection and correction in a tool, CTADetector, on top of Eclipse Java development tools (JDT) [41].

Figure 2.9 shows the architecture of CTADetector. CTADetector takes as input the abstract syntax trees (ASTs) of the analyzed project. It first collects the variables that represent concurrent collections through *Variable Collection* component. In code that use these concurrent collection variables, CTADetector tries to match the code snippets against the check-then-act buggy idioms. If CTADetector finds a match, it reports the buggy instance. CTADetector then corrects each buggy instances through the *Corrector* component, which is an interactive tool.

The process of detecting correct idiom instances is similar to detecting buggy instances. However, in this case, CTADetector matches the code snippets with the correct idioms (similar with the ones shown in Fig. 2.5) rather than the buggy idioms. CTADetector will not invoke *Corrector* component when it detects correct idiom instances.

Figure 2.10(a) shows a sample output for an instance of a detected buggy idiom. When CTADetector finds a match between the source code and an idiom, it reports the detected idiom as well as the source code location. For example, in Fig. 2.10(a), the tool detects six buggy instances, three of which are semantic bugs while others are performance bugs. It also shows the corresponding buggy idiom, file name, and line number of each instance. CTADetector links the report to the source code, so it is easy for the developer to find the location.

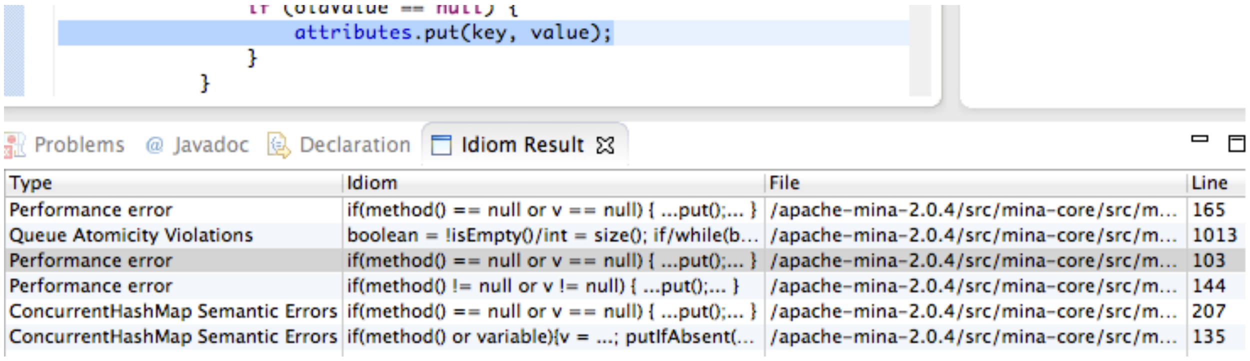
Figure 2.10(b) shows a sample output of the correction. The left pane in Fig. 2.10(b) shows the original buggy code snippet while the right pane shows the code corrected by CTADetector. Notice that this is an interactive process, so the developer can preview and accept/reject the proposed changes.

2.5.2 Idiom detection and correction

To detect idioms, we employ a static code analysis that uses syntactical and semantical information to match conditional statements from the source code of a program to the idioms we presented in Section 2.2.

The analysis visits all the conditional statements (i.e., `if` and `while`) in a program. For each conditional statement, the analysis iterates over all the idioms and tries to determine a match. To determine a match, the analysis needs to verify whether: (i) the conditional expression matches the `check` part of the idiom, (ii) the conditional statement operates over an instance of a concurrent collection, and (iii) the body of the conditional statement matches the `act` part of the idiom.

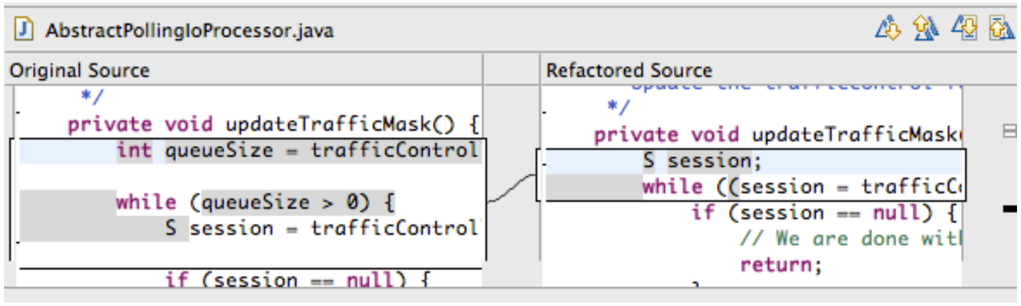
Next, we illustrate how the analysis matches one of the idioms, namely the `put-if-absent`



```
            if (oldValue == null) {
                attributes.put(key, value);
            }
        }
```

Type	Idiom	File	Line
Performance error	if(method() == null or v == null) { ...put();... }	/apache-mina-2.0.4/src/mina-core/src/m...	165
Queue Atomicity Violations	boolean = !isEmpty()/int = size(); if/while(b...	/apache-mina-2.0.4/src/mina-core/src/m...	1013
Performance error	if(method() == null or v == null) { ...put();... }	/apache-mina-2.0.4/src/mina-core/src/m...	103
Performance error	if(method() != null or v != null) { ...put();... }	/apache-mina-2.0.4/src/mina-core/src/m...	144
ConcurrentHashMap Semantic Errors	if(method() == null or v == null) { ...put();... }	/apache-mina-2.0.4/src/mina-core/src/m...	207
ConcurrentHashMap Semantic Errors	if(method() or variable){v = ...; putIfAbsent(...	/apache-mina-2.0.4/src/mina-core/src/m...	135

(a) Output of buggy idiom instances



```
     */
    private void updateTrafficMask() {
        int queueSize = trafficControl

        while (queueSize > 0) {
            S session = trafficControl

            if (session == null) {
```

```
     */
    private void updateTrafficMask
        S session;
        while ((session = trafficC
            if (session == null) {
                // We are done wit
                return;
```

(b) Output of correction

Figure 2.10: A sample output from CTADetector

from Fig. 2.2(e). First, the analysis checks the expression used in the `if`'s condition. This means determining whether (a) the code invokes the `containsKey` (b) the condition is negated.

Second, the analysis checks whether `if` statement operates over an instance of `ConcurrentHashMap`. To do this, the analysis gets the type information of a variable from the static type binding (this determines that the variable is an instance of `Map`) and the variable initialization statement (this determines that the map variable is initialized with a `ConcurrentHashMap`). Note that we use an inter-procedural analysis to find out whether a variable is initialized with a concurrent collection.

Third, the analysis checks whether (a) the body statements invoke the `put` method (b) the `put` is invoked on the same `ConcurrentHashMap` object used in the condition expression, and (c) it places in the map the same `key` object that was used in the condition expression.

Notice that CTADetector uses the same pattern matching approach to detect the

Algorithm 1 $detectCHMPut(ifStatement)$

```
Input: The AST node that represents an if statement.
 1: $expr = ifStatement$.getExpr()
 2: $prefixOperator = expr$.getPrefixOperator()
 3: $thenStmts = ifStatement$.getThenStmts()
 4: if $prefixOperator ==$ Operator.$NOT \wedge expr$.Type == MethodInvocation $\wedge$
          $expr$.invoke() == “containsKey” then
 5:    $chkVar = expr$.getReceiver()
 6:    $chkVType = resolveType(chkVar)$
 7:    $key = expr$.getArgs(0)
 8:    if $chkVType ==$ ConcurrentHashMap then
 9:       for each $stmt \in thenStmts$ do
10:          if $stmt$.Type == MethodInvocation then
11:             $actVar = stmt$.getReceiver()
12:             $actVType = resolveType(chkVar)$
13:             $arg = stmt$.getArgs(0)
14:             if $stmt$.invoke() == “put” $\wedge chkVType == actVType \wedge key == arg$ then
15:                report idiom “if(!map.containsKey(key)) map.put(key,v);”
16:             end if
17:          end if
18:       end for
19:    end if
20: end if
```

correct uses of CHECK-THEN-ACT idioms.

To correct the reported misused idioms, CTADetector uses the fixes that we presented in Subsect. 2.2.8. We implemented the correction on top of Eclipse's AST rewriting engine [42]. Notice that we take an *interactive* approach: the programmer can inspect the report, and if she agrees that it is indeed a problem, she can choose to apply the correction transformation that CTADetector suggests. For each suggested transformation that tool shows a preview of the code before and after the transformation (see Fig. 2.10(b)).

2.5.3 Algorithm for Put-if-Absent Idiom

Algorithm 1 shows the pseudocode for detecting the variation of `put-if-absent` idiom in Fig. 2.2(e). The algorithm takes an `if` statement as the input and gets the expression of the branch condition and the statements in `then` block (lines 1–3). Line 4 examine if the conditional expression satisfies the *check* operation, i.e., there is a negation condition and the invocation of `containsKey` method. Then, the algorithm gets the variable name and type for the variables `map` and `key` (lines 5–7). They are used to examine whether the code snippet operates on a `ConcurrentHashMap` (line 8) and the variables in the *act* part represent the same map and key (line 14). To match the *act* part of the idiom, the algorithm iterates all the statements in the `then` block of the `if` statement (line 9) and try to find a method invocation of `put` (line 14). Next, the algorithm checks whether `put` is called on the same `map`, and same `key` (lines 8–14) as the preceding call of `containsKey`. If all the above information matches, the algorithm reports a buggy instance of the `put-if-absent` idiom.

Due to space considerations, we do not show the pseudocode for the other variations of `put-if-absent` or for other idioms, but they use similar analyses.

2.5.4 Discussion

Despite the fact that our approach is pattern-based, it is quite effective and efficient. We first discuss the reasons for false negatives and false positives and several potential extensions.

1. False negatives: CTADetector only performs an intra-procedural idiom matching, thus it may miss cases when the check and act operations are in different methods (this can lead to false negatives). For example, for the idiom shown in Fig. 2.2(e), the `if(!map.containsKey(key))` and `map.put(key, v)` may be in different methods. However, in the 28 projects we used in our empirical evaluation, we manually found only one single case (in `Mina`) that needs inter-procedural analysis, thus making the inter-procedural analysis unnecessary. This makes sense because the code snippets of check-then-act idioms are succinct, thus the `check` and `act` operations are usually in the same method.

2. False positives: CTADetector uses the static type binding information to determine whether a variable represents a concurrent collection object or whether two arguments are the same. The analysis is a flow-insensitive *may-analysis*. It is conservative and safe but may lead to false positives. Such inaccuracy is intrinsic for static analysis since it only gives an approximation of dynamic execution. Here are two such cases.

First, there might be cases where the analysis determines that the target object on which the CHECK-THEN-ACT idiom is applied *may* point to a `ConcurrentHashMap`, although the object can only point to a `HashMap` in the dynamic execution. In such a case, CTADETECTOR reports a CHECK-THEN-ACT idiom instance on `ConcurrentHashMap`, whereas the idiom instance actually occurs on `HashMap`. However, in the 28 projects we used, there is only one case (in OpenJDK 8) in which a collection may either point to a `HashTable` or a `ConcurrentHashMap`, depending on some conditions.

Second, if the arguments of `check` and `act` operations *may* point to the same objects in the target collection, CTADETECTOR reports an idiom instance. It could happen that the arguments may also be reassigned and point to different objects between the check and act operations, in which case CTADETECTOR reports a false positive. However, we never found such a case in our corpus, because in practice the CHECK-THEN-ACT idioms are succinct.

Thus, our static analysis approach is accurate to detect CHECK-THEN-ACT idioms in most practical cases, and using more accurate flow-sensitive points-to analysis will only have modest improvements.

The last source of imprecision is CTADETECTOR's intra-procedural analysis to determine synchronization. If the idioms are synchronized by locks properly, atomicity violation will not occur. To determine whether a lock protects the idiom, CTADETECTOR checks if the code snippet is in a block protected by a lock and the operations that may result in a atomicity violation are protected by the same lock. Notice that the intra-procedural analysis can miss those cases when the idiom is protected externally by a lock, for example, the caller of the method that contains the idiom acquires a lock before calling the method. However, among the idiom instances we collected, we never found such a scenario, after we checked the code snippets manually.

4. Extensions: To extend CTADETECTOR to detect new idioms, one needs to implement the matching algorithms for new idioms. Suppose that `j.u.c` introduces new kinds of concurrent collections that can have new kinds of CHECK-THEN-ACT idioms. To enable CTADETECTOR to detect these new idioms, one should (i) extend *Variable Collector* to collect variables representing new kinds of collections, (ii) implement the matching algorithms for new CHECK-THEN-ACT idioms, and (iii) extend the *Corrector* component to transform code based on the correct idioms. None of these steps require fundamental conceptual changes to the current algorithms. Because all these steps are similar with what CTADetector already implements, an extender can use the current code as a sample.

CHAPTER 3

Retrofitting Concurrency for Android Applications

3.1 Introduction

According to a Gartner report [43], by 2016 more than 300 billion apps for smartphones and tablets will be downloaded annually. Android is the dominant platform, by 4x over the next largest platform and 1.7x over all the others combined [44]. Previous studies [12, 13] have shown that unresponsiveness is the number one bug that plagues Android apps. To avoid unresponsiveness, app developers use asynchronous programming to execute CPU-bound or blocking I/O operations (e.g., accessing the cloud, database, filesystem) in background and inform the app when the result becomes available.

While programmers can use `java.lang.Thread` to fork concurrent asynchronous execution, it is cumbersome to communicate with the main thread. Android framework provides a better alternative, `AsyncTask`, which is a high-level easy-to-use concurrent construct. However, we know little about how developers use it. To understand how `AsyncTask` is used, underused and misused in practice, we conduct a formative study. The study answers the following questions:

RQ1: *How is `AsyncTask` used?* We found that 48% of the studied projects use `AsyncTask` in 231 different places. Developers either extracted long-running operations into `AsyncTask` via manual refactoring or used `AsyncTask` from the first version.

RQ2: *How is `AsyncTask` misused?* For 4% of the invoked `AsyncTask`, the code runs sequentially instead of concurrently because of invoking wrong APIs. We found similar problems in our previous studies on concurrent libraries in C# [45, 46]. On the other hand, in 13 cases, code in `AsyncTask` accesses non thread-safe GUI widgets. This leads to data races on these GUI widgets.

RQ3: *Is `AsyncTask` underused?* We found that 251 places in 51 projects execute long-running operations in UI event thread. This also confirms the findings of a recent study by Liu et al. [12] that shows that 21% of reported responsiveness bugs in an Android corpus arise because developers tend to forget encapsulating long-running operations in `AsyncTask`.

Inspired by these findings, we designed, developed, and evaluated ASYNCHRONIZER, an automated refactoring tool that enables developers to extract long-running operations into `AsyncTask`. ASYNCHRONIZER uses a points-to static analysis to determine the safety of the transformation.

3.2 Background on Android AsyncTask

3.2.1 Android GUI Programming

```
public class MainActivity extends Activity {
    public boolean onOptionsItemSelected(MenuItem item) {
        ...
        new Button(new OnClickListener() {
            public void onClick(...) {
                exportToSpreadsheet(gameIds);
            }
        });
    }
    private void exportToSpreadsheet(final List gameIds) {
        ...
        new AsyncTask<Void, Void, String>(){
            protected void onPreExecute(String filename) { }
            protected String doInBackground(Void... params) {
                ...
                for (Object gameId : gameIds) {
                    games.add(dbHelper.findGameById(gameId));
                    publishProgress((Void)null);
                }
                String filename = ...
                return filename;
            }
            protected void onProgressUpdate(Void... values) {
                progressDialog.incrementProgressBy(1);
            }
            protected void onPostExecute(String filename) {
                progressDialog.dismiss();
            }
            protected void onCancelled (String filename) { }
        }.execute((Void)null);
    }
}
```

Figure 3.1: Real-world example of `AsyncTask` in `KeepScore`

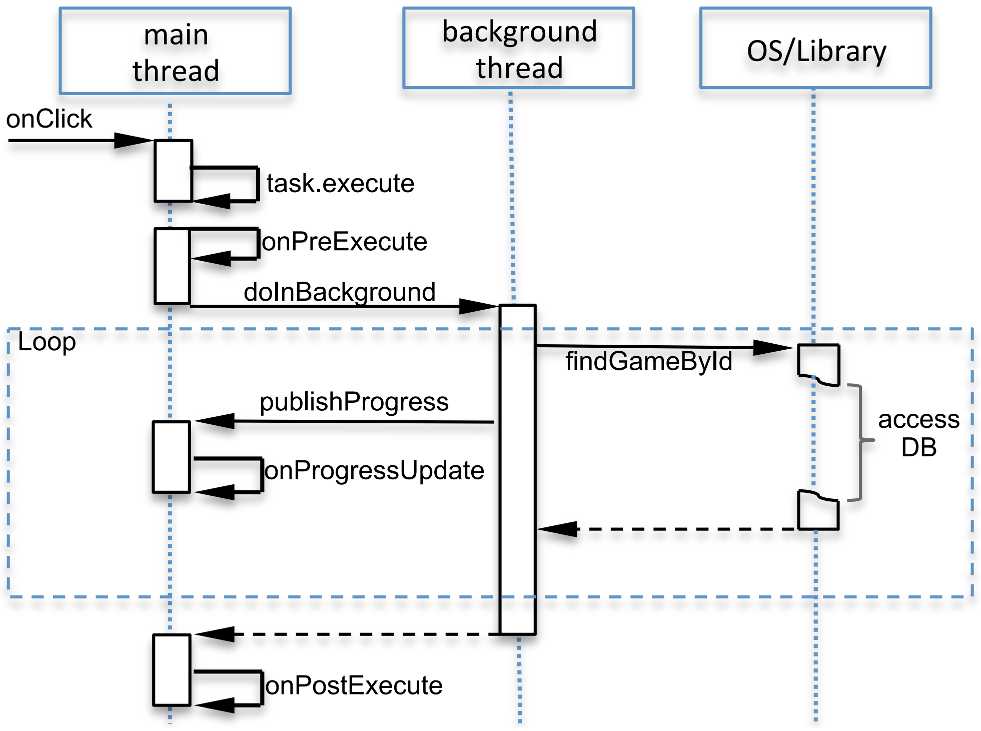


Figure 3.2: Where is `AsyncTask` code executing?

Android GUIs are typically composed of several *activities*. An activity represents a GUI screen. For example, the login screen of an email client is an activity. An application GUI transitions through a sequence of activities, each of which is independent of the others. However, at any given time, only one activity is active. Activities contain GUI widgets.

Similar to many other GUI frameworks such as Swing [47] and SWT [48], Android uses an event-driven model. Events in Android apps include lifecycle events (e.g., activity creation), user actions (e.g., button click, menu selection), sensor inputs (e.g., GPS, orientation change), etc. Developers define event handlers to respond to these events. For example, `onCreate` handler is invoked when an activity is created, while `onClick` handler of a button is invoked when a button is clicked.

Android framework uses a **single thread model** to process events [49]. When an application is launched, the system creates a *main thread*, i.e., the *UI event thread*, in which it will run the application. This thread is in charge of dispatching UI events to appropriate widgets or lifecycle events to activities. The main thread puts events into a single event queue, dequeues events, and executes event handlers.

However, if the main thread executes CPU-intensive work or blocking I/O calls such as network access or database queries, this results in poor responsiveness. Once the main thread is blocked, no events can be dispatched and processed, so application will not be responsive to users' actions. To avoid unresponsiveness, developers should exploit concurrency and extract long-running operations into another thread.

3.2.2 AsyncTask in Android Framework

To ease the use of concurrency, Android framework provides `AsyncTask` class. `AsyncTask` is a high-level abstraction for encapsulating concurrent work. `AsyncTask` also provides event handlers such as `onPostExecute` that execute on the main thread after the task has finished. Thus, the background task can communicate with the main thread via these event handlers.

We illustrate a typical usage of `AsyncTask` using a real-world app, `KeepScore`, that keeps scores for games that require tallying points as one plays. Figure 3.1 shows an `AsyncTask` that reads game scores from a database and exports them in a spreadsheet file. The methods that start with "on" are event handlers. Figure 3.2 shows the flow of this `AsyncTask`.

Line 4 sets up a listener for a button, and when the button is clicked, method `exportToSp-readsheet` is called. This method creates an `AsyncTask` (line 12) and executes it concurrently with the main thread. The `doInBackground` method (line 14) encapsulates the work that executes in the background. The task queries a database and adds the results to a list, `games` (line 17). Finally, the result of the background computation is returned for main thread to use (i.e., `filename` at line 21).

While the task is executing, it can report its progress to the main thread by invoking `publishProgress` and implementing `onProgressUpdate` handler. In the example, the task publishes its progress every time it finds a game (line 18), so the main thread can update a progress dialog (line 24). The main thread executes the `onPostExecute` handler after `doInBackground` finishes. In this example, the handler updates the GUI by dismissing the progress dialog (line 27). Notice that this handler takes the result of the task as parameter (`filename` at line 26).

When it manages the lifecycle of an `AsyncTask`, the main thread executes `onPreExecute` (line 13) before the `doInBackground`. It also executes the `onCancelled` (line 29) when the task is cancelled.

The three generic types of `AsyncTask` (line 12) represent the parameter types of `doInBac-kground`, `onProgressUpdate`, and the return type of `doInBackground`.

Notice that there are two ways that the main thread can fetch the result of an `AsyncTask`. One, the result is available in the `onPostExecute`. Second, the result can be explicitly requested through the `get` method on the task. This method has blocking semantics: if the result is available, it will return immediately, otherwise it will block the main thread until the result becomes available.

3.3 Formative Study of AsyncTask Use

In this section we present our formative study to understand how developers use, misuse, and underuse `AsyncTask` in open-source Android apps.

3.3.1 Experimental Setup

Corpus of Android Apps. We selected our corpus of Android apps from Github [50]. To find Android apps, we filter Java repositories by searching if their `README` file contains "Android app" keyword. We also manually confirmed that these repositories are Android apps. We apply two more filters. First, because we want to contact developers, we need to avoid analyzing inactive projects. Thus, we only keep repositories that contain at least one commit after June 2013. Second, because we want to study the usage of `AsyncTask` in mature, representative apps, we ignore apps that have less than 500 SLOC. Also, we ignore all forked applications since they are similar to the original repository. Finally, we use the top 104 most popular projects as our corpus, comprising 1.34M SLOC, produced by 1139 developers.

Analysis. We want to study whether developers refactor existing code into `AsyncTask` (i.e., they encapsulate existing statements into `AsyncTask`), or whether they introduce `AsyncTask` on new code they write from scratch. Thus, we study not the latest version of the code which contains `AsyncTask`, but the first version of the code where developers introduce this construct. To do this, we searched the commit history of our corpus through GiTective API [51], identified the commits that add import statements to `AsyncTask`, and manually inspected the versions before and after such commits. This helps us understand questions about correct and incorrect usage.

To understand whether the corpus contains underusage of `AsyncTask`, we want to identify long-running operations that execute in the UI event thread and are potentially decreasing the responsiveness of the application. These operations are candidates to be encapsulated within `AsyncTask`.

Thus we first created a "black list" of long-running operations that Android documentation [49] warns should be avoided in the UI. We used Eclipse's search engine to find call sites to such operations. Using the call graph hierarchy, we analyzed whether they appear directly or indirectly in event handlers but not in `AsyncTask` or `Thread`.

To assure validity and reliability, we make all the data-set and the results available online [52].

3.3.2 Results

Table 3.1 shows the results about usage and misusage in the 50 projects that use `AsyncTask`. The second row shows the items we count, including the number of instances of `AsyncTask` (column 2), how many event handlers of `AsyncTask` are implemented by developers (columns 3 to 6), number of misuse which includes accessing GUI in `doInBackground` (column 7) and wrong usage of `get` (column 8). The third row counts these items in newly introduced `AsyncTask` (i.e., code where developers use `AsyncTask` from scratch). The fourth row counts these items in code that was manually refactored by developers to use `AsyncTask`. The fifth row sums the usage in newly introduced and refactored `AsyncTask`.

Using the data in Tab. 3.1, we answer three questions:

RQ1: What are the commonly used CHECK-THEN-ACT idioms in real-world programs?

Table 3.1: AsyncTask usage and misuage.

Item	Usage					Misusage	
	AsyncTask instances	onPost-Execute	onPre-Execute	onProgre-ssUpdate	onCancelled	GUI access	Wrong usage of get
Newly intro-duced	125	123	44	7	6	6	0
Manually Refac-tored	106	64	10	2	0	7	9
Total	231	187	54	9	6	13	9

50 out of 104 projects use AsyncTask to embrace concurrency. This shows AsyncTask is adopted in Android apps.

54% (125 out of 231) of AsyncTask instances are newly introduced when developers add new features. However, there are 46% (106 out of 231) AsyncTask refactored. Here we found two refactoring scenarios. First, in 94 cases, the code was refactored directly into AsyncTask. Second, in 12 cases, the code is refactored from Java Thread into AsyncTask. This is reasonable since AsyncTask provides event handlers and is easier to use than Thread when the background thread needs to communicate with the main thread.

Lastly, we notice that onPostExecute handler is the most widely implemented by developers (81% (187 out of 231)). However, for the other three handlers, the implementation percentage is only 23%, 4% and 3%. A possible explanation is that after a task is done, applications have to update UI and report the result to users. onPostExecute handler provides an easy way to update UI without explicitly knowing when the task is finished, so it is implemented in most cases.

RQ2: Which idioms are the most error-prone?

We found that 13 AsyncTask (7 in manual refactoring) access GUI in doInBackground. However, based on the Android document, accessing GUI from outside main thread will lead to races, because Android UI toolkit is not thread-safe.

Data races can also occur on the non-GUI objects after developers transform sequential

```
 1 class NewChatActivity extends SherlockFragmentActivity {
 2     public void onLoadFinished(Loader loader, Cursor cursor){
 3         resolveIntent();
 4         if (mRequestedChatId >= 0) {
 5             ...
 6         }
 7     }
 8     private void resolveIntent() {
 9         startGroupChat(path, host, listConns.get(0));
10         ...
11     }
12     private void startGroupChat(...) {
13         ...
14         new AsyncTask<String, Void, String>() {
15             protected String doInBackground(String... params){
16                 ...
17                 mRequestedChatId = session.getId();
18             }
19         }.execute(room, server);
20     }
21 }
```

Figure 3.3: In `ChatSecureAndroid` project, developers introduce races in manual refactoring.

code to concurrent code. Figure 3.3 shows a manual refactoring in `ChatSecureAndroid` project. At line 3, `onLoadFinished` handler eventually calls `startGroupChat` method, in which an `AsyncTask` is executed. This task writes to field `mRequestedChatId` at line 17. However, this field is read at line 4, which can be executed concurrently with line 17. Thus, there is a race on `mRequestedChatId`. Note that this data race is found by Asynchronizer in our evaluation (see Sec. 3.6) rather than being found manually in this formative study.

Also, nine manually refactored `AsyncTasks` are misused because developers invoke `get` method on the task immediately after starting the task. As we mentioned in Sec. 3.2, invocation of `get` blocks the current thread until the result is available. Thus, such usage blocks the main thread immediately and defies the purpose of using `AsyncTask`.

RQ3: Do misused idioms result in real bugs? Are our patches accepted by developers? We found that 51 out of 104 projects call long-running APIs in UI event handlers at 251 places. In these 51 projects, 17 projects have already used `AsyncTask`. Still, we found 79 places where `AsyncTask` is underused. The remaining 34 projects never use `AsyncTask`.

Based on our findings for RQ1, we conclude that `AsyncTask` is widely adopted and developers manually refactor their code to use `AsyncTask` in many cases. However, as RQ3 shows, `AsyncTask` is still underused. RQ2 shows that manual refactoring may introduce bugs.

Based on the results for RQ1–RQ3, we conclude that there is a need for safe refactoring tools to enable developers to transform code towards `AsyncTask` (presented in Sec. 3.4), as well as help developers check possible races that can occur between the code running in `AsyncTask` and the code running in the main thread (presented in Sec. 3.5).

3.4 Transformations

```
 1 public class RouteselectActivity extends Activity {
 2   ...
 3   public void onCreate(Bundle savedInstanceState) {
 4     super.onCreate(savedInstanceState);
 5     ...
 6     ListView lv = getListView();
 7     final String qry = "select...";
 8     final String[] selectargs = {mStopid, datenow,
 9        datenow};
10
11
12
13
14
15
16
17
18
19
20
21
22     Cursor mCsr = DatabaseHelper.ReadableDB()
23         .rawQuery(qry, selectargs);
24     startManagingCursor(mCsr);
25
26
27
28     lv.setOnTouchListener(mGestureListener);
29     if (mCsr.getCount() > 1)
30       tv.setText(R.string.route_fling);
31     else if (mCsr.getCount() == 0)
32       tv.setText(R.string.stop_unused);
33     lv.addFooterView(tv);
34       CursorAdapter adapter =
35          new CursorAdapter(this,
36          mCsr);
37     setListAdapter(adapter);
38   }
39
40 }
```

(a) before

```
 1 public class RouteselectActivity extends Activity{
 2   ...
 3   public void onCreate(Bundle savedInstanceState){
 4     super.onCreate(savedInstanceState);
 5     ...
 6     ListView lv = getListView();
 7     final String qry = "select...";
 8     final String[] selectargs = {mStopid, datenow,
 9        datenow};
10     ProcessRoutes prTask = new ProcessRoutes(lv);
11     prTask.execute(qry, selectargs);
12   }
13   class ProcessRoutes extends AsyncTask<Object,
14        Void, Cursor>{
15     ListView lv;
16     ProcessRoutes(ListView lv) {
17       this.lv = lv;
18     }
19     protected Cursor doInBackground(Object... args){
20       String qry = (String) args[0];
21       String[] selectargs = (String[]) args[1];
22       Cursor mCsr = DatabaseHelper.ReadableDB()
23          .rawQuery(qry, selectargs);
24       startManagingCursor(mCsr);
25       return mCsr;
26     }
27     protected void onPostExecute(Cursor mCsr) {
28       lv.setOnTouchListener(mGestureListener);
29       if (mCsr.getCount()>1)
30         tv.setText(R.string.route_fling);
31       else if (mCsr.getCount() == 0)
32         tv.setText(R.string.stop_unused);
33       lv.addFooterView(tv);
34       CursorAdapter adapter =
35          new CursorAdapter(RouteselectActivity.this,
36          mCsr);
37       setListAdapter(adapter);
38     }
39   }
40 }
```

(b) after

*Relevant code from GR-Transit app. Programmer selects lines 22 to 24 in (a), and ASYNCHRONIZER performs all the transformations. The left-hand side shows the original code, whereas the right-hand side shows the refactored code by ASYNCHRONIZER.

Figure 3.4: An example refactoring performed by ASYNCHRONIZER.

This section describes the code transformation that enables developers to move code from main thread into `AsyncTask`. We implement the transformation in our tool, ASYNCHRONIZER. We first explain the overall workflow of the tool, and then illustrate the transformations.

3.4.1 Refactoring Workflow and Preconditions

We implement ASYNCHRONIZER as a plugin in the Eclipse IDE [42]. To use ASYNCHRONIZER, the programmer selects statements that she wants to encapsulate within `AsyncTask`, and then chooses the CONVERT TO ASYNCTASK option from the refactoring menu. The programmer can also specify the class and instance name that ASYNCHRONIZER will use to generate `AsyncTask`. ASYNCHRONIZER moves the selected statements into `AsyncTask.doInBackground` method. In addition, ASYNCHRONIZER also infers the subsequent statements that can be moved to `onPostExecute`. Before applying the changes, ASYNCHRONIZER gives the programmer the option to preview them in a before-and-after pane. Then, ASYNCHRONIZER transforms the code in place.

After the transformation, the programmer can invoke ASYNCHRONIZER's safety analysis component to check data races due to the transformation. We will present the safety analysis in Sec. 3.5. If ASYNCHRONIZER found data races, the programmer still needs to confirm and fix them manually. Only after this the refactored code is correct.

Figure 3.4(a) shows a code snippet from an Android app, `GR-Transit`, that displays bus routes and schedules . The code snippet is used to show the bus routes that pass a given bus stop. If the programmer applies our transformation to lines 22 to 24, ASYNCHRONIZER will transform the code to Fig. 3.4(b). In a subsequent version of `GR-Transit`, the programmers have done this transformation manually. Their new code, modulo syntactic difference, is the same as ASYNCHRONIZER's output.

ASYNCHRONIZER checks the following three preconditions before transforming, and reports failed preconditions:

(P1) The selected statements do not write to more than one variable which is read in the statements after the selection. Such a variable needs to be returned by `doInBackground`, but Java methods can only return one single variable.

(P2) The selected statements should not contain `return` statements. A `return` statement in the original code enforces an exit point from the enclosing method. However, the same `return` statement extracted into an `AsyncTask` can no longer stop the execution of the original method. Similarly, the `break` and `continue` statements are only allowed if they are selected along with their enclosing loop.

(P3) The selection contains only entire statements. Selecting an expression that is part of a statement is not allowed because it would force the `AsyncTask` to immediately invoke the blocking `AsyncTask.get()` to fetch the expression; this defies the whole purpose of launching an `AsyncTask`.

3.4.2 Create the doInBackground Method

The first step of the transformation is to move the selected statements into the `doInBackgro-und` method. This is similar to EXTRACT METHOD refactoring. In this step, ASYNCHRONIZER needs to determine the arguments and the return value of `doInBackground`. The arguments are the local variables which are used in the selection but declared before it. The return value is the local variable which is defined in the selection but used after it.

In Fig. 3.4(b), the `doInBackground` method takes two arguments, `qry` and `selectargs`, and returns `mCsr`. However, note that `doInBackground` has only one *varargs* parameter (i.e., array of unknown length), and its type is specified by the type argument (generic) of `AsyncTask`. If all local variables are of the same type, ASYNCHRONIZER sets this type as the first generic type argument for `AsyncTask`. If the passed-in local variables are of different types, as it is the case for our example, ASYNCHRONIZER uses `java.lang.Object` as the generic type argument (Fig. 3.4(b) line 19), and dereferences and type-casts the parameters (Fig. 3.4(b) lines 20 and 21). If `doInBackground` has no arguments or return value, it uses `Void` as parameter type or return type, and returns `null`.

Algorithm 2 $inferringPostStmts(selected, post, rv)$

```
Input: selected ← the selected code
       post ← all statements after the selected code
       rv ← return variable of doInBackground
Output: moved ← statements put into onPostExecute
 1: dominated ← []
 2: unmoved ← []
 3: for all stmt in post do
 4:     if selected dominates stmt and not stmt contains return then
 5:         dominated ← dominated append stmt
 6:     else
 7:         break
 8:     end if
 9: end for
10: unmoved ← post − dominated
11: moved ← []
12: for all stmt in dominated do
13:     if not unmoved is data dependent on stmt then
14:         moved ← moved append stmt
15:     else
16:         break
17:     end if
18: end for
19: unmoved ← post − moved
20: if unmoved uses rv then
21:     invoke get method before the first use of rv in unmoved
22: end if
23: return moved
```

3.4.3 Create onPostExecute Handler

The second step is to infer which code can be put into `onPostExecute` handler. Because the Android framework invokes the `onPostExecute` after the method `doInBackground` has finished, the analysis needs to determine that the statements inside these two methods follow the same control-flow as in the original program. Otherwise, the refactored program will have a different semantics.

A naive implementation is to move all the statements after the selected code into `onPostExecute`. However, this may break the control flow of the main thread. A statement cannot be moved if it is not dominated by the statements in the selected code, or if it is a `return` statement. A statement *dominates* [53] another if every path from the entry point to the latter statement passes through the former statement.

Algorithm 2 infers the set of statements to be moved to `onPostExecute`. The inputs of the algorithm include the selected code that will be put into `doInBackground` (*selected*), the list of statements syntactically after the selected code (*post*), and the return variable of `doInBackground` (*rv*). The output is the set of statements which can be moved to `onPostExecute` (*moved*). *unmoved* contains the statements which cannot be moved.

The algorithm first selects the prefix of *post* in which all statements are dominated by *selected* and do not `return` (lines 3 to 9). The remaining statements cannot be moved so they are put into the *unmoved* variable (line 10). The algorithm then constructs the final result as the prefix of *dominated* for which all statements have no effect on any statement in *dominated* (lines 11 to 18). This ensures no data dependencies are broken. *unmoved* is updated with any statements which are not in *moved* (line 19). Finally, if *unmoved* contains statements that use the resulting value of `doInBackground`, ASYNCHRONIZER adds a call to `AsyncTask`'s `get` method before the first such use (lines 20 to 22).

In the example shown in Fig. 3.4(a), all the statements after the selected code (lines 28 to 37) can be put into `onPostExecute`. However, suppose there was a statement at line 38 that returns `mCsr`. This statement would not be moved to `onPostExecute`. Furthermore, ASYNCHRONIZER would add a call to `AsyncTask.get` before the return statement because it uses `mCsr`. In the current implementation, ASYNCHRONIZER uses Eclipse JDT's [41] variable bindings to approximate data dependencies.

3.4.4 Create Class Declaration

In this step, ASYNCHRONIZER creates fields, constructor and class declaration for `AsyncTask`. Fields are generated by analyzing the statements in `onPostExecute`. Since `onPostExecute` only have one parameter which is the return value of `doInBackground`, the tool converts all the other arguments needed by `onPostExecute` into fields of `AsyncTask`. For example, in Fig. 3.4(b), local variable `lv` is needed by `onPostExecute`. ASYNCHRONIZER declares a field `lv` in the `AsyncTask` (line 15) and adds a constructor to initialize this field (line 16).

After that, it creates an inner class declaration using all the code elements which have been created above (line 13). Finally, it generates two statements to create task instance and call `execute` method, and replaces the selected code by these two statements (lines 10 and 11).

3.4.5 Special Cases

ASYNCHRONIZER also analyzes code to properly transform several special cases:

(S1) `doInBackground` and `onPostExecute` cannot be declared to throw checked exceptions. Thus, if the selected statements throw exceptions (e.g., programmer selects `FileOutputStream.write` method which throws `IOException`), ASYNCHRONIZER needs to generate try-catch block to handle the exceptions. ASYNCHRONIZER first collects the exceptions that are declared to be thrown by the selected code. If these exceptions are caught in the original refactored method, it copies the corresponding catch clauses into `doInBackground` or `onPostExecute` to handle the exceptions. Otherwise, it generates empty catch clause. In our experiment, all the cases that throw exceptions have corresponding catch clauses in the original code.

(S2) The original code may use `this` or `super` pointer (Fig. 3.4(a) line 35). After moving it to an inner `AsyncTask` class, our tool replaces the original pointer with outer class' `this` or `super` pointer (Fig. 3.4(b) line 35).

3.5 Data Race Check

Our formative study (Sec. 3.3) shows that developers do introduce data races when they manually refactor sequential code into `AsyncTask` concurrent code. These data races are either accesses to GUI elements from the `doInBackground`, or possibly concurrent accesses to other shared resources. Data races are hard to find as they only manifest themselves under certain thread schedules. To assist developers with the refactoring, we propose a static race detection approach specialized to the thread structure generated by `AsyncTask`. We implement our approach as an extension of the ITERACE race detector [54].

ITERACE is a static race detector for Java parallel loops that achieves low rates of false warnings by taking advantage of the extra semantic information provided by the use of high-level concurrency constructs. It uses the known thread-safety properties of concurrent collection classes, summarizes races that occur in libraries at the level of the application, and specializes for the thread structure of lambda-style parallel loops.

While IteRace is only capable of analyzing parallel loops, its approach of taking advantage of the implicit thread structure of high-level concurrency constructs is also applicable to `AsyncTask`. We thus extend IteRace to find races that occur between `doInBackground` and other threads.

3.5.1 Data Races

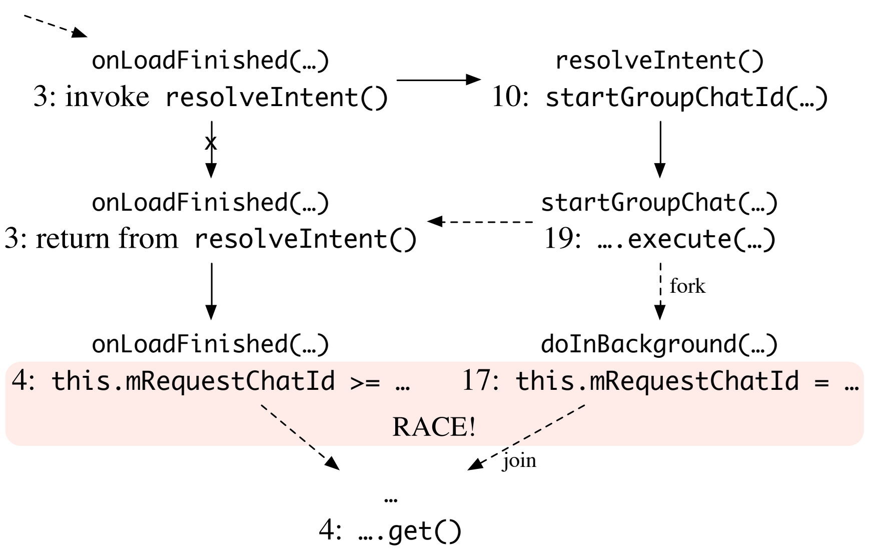
Generally, a data race is a pair of accesses, one of which is a write, to the same memory location, with no ordering constraint between the two accesses. For `AsyncTask`, a data race can occur between accesses in `doInBackground` and accesses which may execute in parallel with the asynchronous task. While the precise set of instructions that may execute in parallel cannot be determined statically, we can find an approximation of it.

Asynchronizer relies on the Andersen-style static pointer analysis [55] provided by WALA [56]. Thus, our analysis works over an abstract heap built along with a (k-bounded) context-sensitive call graph. The underlying analysis is flow insensitive, except for the limited sensitivity obtained from the SSA form.

Our tool makes the following approximation for a race: instruction i_α in call graph node n_α races with instruction i_β in node n_β if both access the same field of the same abstract object, at least one of the instructions is a write access, and $\langle n_\alpha, i_\alpha \rangle$ may happen in parallel with $\langle n_\beta, i_\beta \rangle$.

3.5.2 May Happen in Parallel

We now introduce an approximation of the happens-in-parallel relation induced by the `AsyncTask`. For simplicity, we present the algorithm from the perspective of analyzing the races involving one `AsyncTask` at a time.



*The nodes are call graph node-instruction pairs. The arrows are intra and inter procedural edges. The crossed-out arrow is part of G^* but not $G^*_{c \not\to r}$. Dashed arrows denote *reachable* relations.

Figure 3.5: Supergraph without call-to-return edges ($G^*_{c \not\to r}$) for the code snippet in Fig. 3.3.

Let n_b be the abstract call graph node for the analyzed `doInBackground` method. Let n_h be the event handler call graph node which executed n_b's `AsyncTask`– note that, depending on the choice of abstraction, there can be multiple call graph nodes representing runtime invocations of `doInBackground`. Let N_h be the set of all the event handler call graph nodes in the current application. For the example in Fig. 3.3, n_b is the invocation of the `doInBackground` method on line 15, and n_h is the execution of `onLoadFinished` (line 2) which led to n_b.

Let i_e be the instruction which executes the `AsyncTask` containing n_b. For our example in Fig. 3.3, i_e is `execute` method invocation at line 19. Let n_e be the node executing i_e. Our choice of context sensitivity ensures its uniqueness.

Let G^* be the so called supergraph [57] having as nodes pairs $\langle n, i \rangle$, where n is an call graph node, and i is an instruction in n. Intra-procedural, i.e., control flow graph (CFG), nodes and edges are lifted to the new graph, with each node i becoming a pair $\langle n, i \rangle$ and each edge $\langle i_1, i_2 \rangle$ becoming $\langle \langle n, i_1 \rangle, \langle n, i_2 \rangle \rangle$. Call sites are linked to the lifted CFG of the target call graph node. The call site instruction is represented by two instructions in G^*, a *call* and a *return*. The call instruction is linked to the entry of the lifted CFG of the target CG node, while the return instruction is linked from the exit. Finally, there is an intraprocedural edge, *call-to-return*, which bypasses the interprocedural paths by linking the call and the return instructions directly.

Let $G^*_{c \not\to r}$ be G^* with all its call-to-return edges removed. Figure 3.5 shows the supergraph without call-to-return edges for the example in Fig. 3.3. Removing call-to-return edges does not affect reachability but it does affect the dominator relation used below. Call-to-return edges prevent instructions in a called method dominate any instruction after the call.

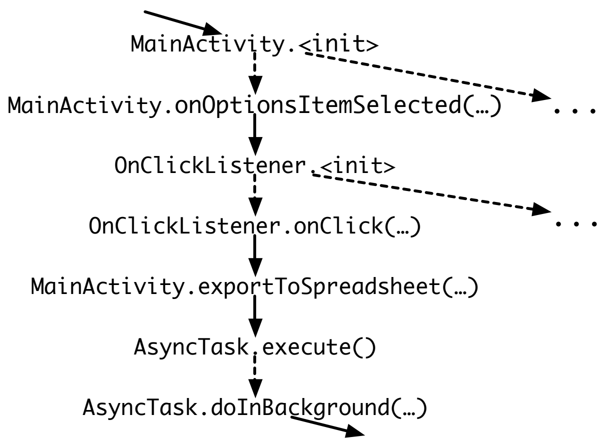
We say that the instruction $\langle n_\alpha, i_\alpha \rangle$ *may happen in parallel* with instruction $\langle n_\beta, i_\beta \rangle$ if $\langle n_\alpha, i_\alpha \rangle$ is reachable from the `doInBackground` node n_b, and either $\langle n_\beta, i_\beta \rangle$ does not dominate $\langle n_e, i_e \rangle$ on $G^*_{c \not\to r}$, or, if n_β calls n_e, i_β does not dominate the call to n_e on the n_β's CFG. E.g., in Fig. 3.5, $\langle n_\alpha, i_\alpha \rangle$ is the node for line 17 which is within the `doInBackground` method. $\langle n_\beta, i_\beta \rangle$ is the node for line 4, which does not dominate the forking node (line 19). Thus, the nodes for lines 4 and 17 may happen in parallel.

Furthermore, as the two instructions read and write the same field (`mRequestChatId`) of the same object (`this`), they may race. Thus, our tool raises a warning.

3.5.3 Android Model

Android applications are event-based so exercising the code depends on events triggered by the UI, sensors, network, etc. In order to analyze the application statically, ASYNCHRONIZER uses a synthetic model of several key Android classes.

Figure 3.6 shows the callgraph for the code snippet in Fig. 3.1. ASYNCHRONIZER creates synthetic calls between the object initializer (the bytecode `<init>` method called before the constructor) of an activity or widget and its events handlers. Thus, `MainActivity`'s initializer calls, among others, its `onOptionsSelected` event handler. Similarly, ASYNCHRONIZER puts a synthetic call between a listener's initializer node to its handlers, and between the an `AsyncTask`'s `execute` and its `doInBackground`. This is an over-approximation of the application's possible behavior because it may be possible that a particular event will not be triggered. As the analysis is flow insensitive, it does not matter that the handler method is invoked at the handler object initialization point, not at the event trigger point.


MainActivity.<init>
MainActivity.onOptionsItemSelected(…)
OnClickListener.<init>
OnClickListener.onClick(…)
MainActivity.exportToSpreadsheet(…)
AsyncTask.execute()
AsyncTask.doInBackground(…)
. . .
. . .

*Dashed arrows are synthetic call graph edges.

Figure 3.6: Part of the callgraph for the code in Fig. 3.1.

ASYNCHRONIZER use the following strategy to select entry point for the analysis: *(1)* if the refactored class itself is an activity, it uses `Activity.<init>` as entry point; *(2)* if the refactored class is a GUI widget class (i.e., a `View`), it uses the object initializer of both the activities who use this widget, and the widget class itself as entry point (i.e., the analysis may run multiple times).

In terms of safety, our analysis is subject to the traditional limitations of static pointer analysis. Aside from the synthetic calls described above, reflection and native code are only handled up to what is provided by the underlying pointer analysis engine, WALA [56]. In some cases, Android apps use reflection to construct objects such as GUI widgets. Our analysis does not analyze such objects. This could be improved by looking into the configuration files used for defining the UI [58]. Also, the analysis' call graph contains a single node for each event. Considering our may happen in parallel definition, this may lead to false negatives for the cases where an event is invoked repeatedly.

Regarding precision, as our race detection analysis is static, it may report false races. The imprecision stems from various types of imprecision in the underlying pointer analysis. This is currently an unavoidable problem for scalable static race detectors[59, 54, 60]. First, the pointer analysis may abstract multiple runtime objects by a single abstract object, leading to false warnings on fields of objects that are always distinct at runtime. Second, the analysis is flow-insensitive leading to warnings between accesses that are always ordered at runtime. In particular, our current implementation does not consider event handling order. This leads to some false warnings in our evaluation. For example, the `onCreate` handler is always handled before `onStart`. Thus, an `AsyncTask` started in `onStart` could not happen in parallel with `onCreate`.

3.6 Evaluation

To evaluate the usefulness of ASYNCHRONIZER we answer the following evaluation questions:
EQ1. Applicability: How applicable is ASYNCHRONIZER?
EQ2. Accuracy: How accurate is ASYNCHRONIZER when performing the code transformations?
EQ3. Effort: How much programmer effort is saved by ASYNCHRONIZER?
EQ4. Safety: Is ASYNCHRONIZER safer than manual refactorings?
EQ5. Value: Do programmers find refactorings applied by ASYNCHRONIZER useful?

3.6.1 Experimental Setup

We want to evaluate ASYNCHRONIZER on real open-source code, but because we are not the original developers of the code, it is hard to know on which code to apply the refactoring. Thus, we use two sets of experiments. First, we let the source code itself tell us which parts need to be refactored. To do this, we run ASYNCHRONIZER on projects that were manually refactored by the open-source developers, and compare the outcomes. Second, we start from the responsiveness issues detected by other researchers [13] and run ASYNCHRONIZER on code that was not refactored yet and determine whether the refactorings are useful for the original code developers. We use the first experiment to answer EQ1–EQ4, and the second experiment to answer EQ5.
Replicating existing refactorings. From our formative corpus of 104 projects, we filtered all projects which have at least two manual refactorings from sequential code to concurrent code via `AsyncTask`, thus resulting in a corpus of 13 projects. The left-hand side of Tab. 3.2

shows the size of each project in non-blank, non-comment source lines of code[1]. For each project, we applied Asynchronizer to the code version just before the version that contained manual refactorings, and we only refactored the same code as the manual refactorings did. Notice that manual refactorings occur in several versions, so we checked out the version we need every time we applied Asynchronizer. We applied Asynchronizer to replicate all 77 manual refactorings in these 13 projects.

We report several metrics for each project. To measure the applicability, we count how many code fragments met the refactoring preconditions and thus can be refactored.

To measure the accuracy of code transformations, we compared the code transformed by Asynchronizer with manually changed code, and report the number of cases that have differences in `doInBackground` or `onPostExecute` method. Notice that here we are only interested to compare the code changes (described in Sec. 3.4), but these changes may still contain data races (we answer safety separately).

To measure the effort that a programmer would spend to refactor the code manually, we report the number of lines and files that are modified by the refactoring. These numbers are a gross estimate of the programmer effort that is saved when refactoring with Asynchronizer. Although we measure effort indirectly, many changes and analysis are non-trivial.

To answer the safety question, we ran Asynchronizer to analyze data races introduced by transformation. Notice that the races which occur in libraries (e.g., JDK) are not reported at that level, but Asynchronizer propagates the accesses up the call graph to the places where the library is invoked from the application [54]. We manually checked all the races and categorize them into four categories: *(fixed directly)* the races are fixed by developers during their manual refactoring; *(fixed later)* the races are not fixed during manual refactoring, but are fixed in a later version; *(never fixed)* the races are not fixed even in the latest version; *(false)* the races are false warnings.

The races that are fixed directly manifested immediately after a developer first encapsulated code into `AsyncTask`. Since in their commit the developers included both the refactoring and the race fixes, it implies that they are aware of the existence of these races. For the races that are fixed later, we also count how many days on average it took developers

[1]We used David Wheeler's SLOCCount [39] to get size and we only report size of Java code.

to find and fix races, as reported by the time span between the commit that introduces the race and the commit that fixes the races. For the races that are still not fixed in the latest version, we reported all of them to developers and suggested how to fix them.

Applying new refactorings. The preferred way to test for responsiveness is to run performance tests. However, none of the Android apps that we found had performance tests. Creating performance tests requires domain knowledge, generating test inputs that are representative (e.g., relevant database entries), etc. Thus, to measure the value of the refactoring, we select 19 projects that have potential responsiveness issues (shown in Tab. 3.3). These issues are either detected in [13] or in our study presented in Sec. 3.3 but they have neither been reported, nor fixed.

We manually identified the latent long-running operations in main thread. For example, we search for call sites to database APIs in main thread. Then, we applied Asynchronizer on these operations and generated patches from the refactoring. When Asynchronizer raised a race warning, we checked and fixed the race. We also included the fix in the refactoring patches. We submitted these patches to developers. In total, we applied Asynchronizer to 123 places (column `Passed` + `Failed` in Tab. 3.3) in these projects. We grouped all changes and submitted 19 patches (one patch per project).

3.6.2 Results

Table 3.2 tabulates results of applying Asynchronizer to 13 projects that have manual refactorings.

Applicability: Columns 3 and 4 show the number of refactorings that pass or fail preconditions P1–P3. Among the 77 places where we applied the refactoring, 73 places satisfy all the three preconditions. Thus, our refactoring is highly applicable.

Of the four places that fail preconditions, 3 failed P1, 1 failed P2, 1 failed P3 (one case failed two preconditions). We had to manually modify the code to satisfy the preconditions. To satisfy precondition P1, we convert the local variables into fields of the refactored class. For precondition P2, we temporarily remove the return statements before refactoring and put them back to the appropriate places after refactoring. For precondition P3, we

Table 3.2: Results of applying ASYNCHRONIZER to 13 projects that have manual refactorings.

Project Name	SLOC	Applicability		Accuracy	Effort		Safety					
		Passed	Failed	Diff. #	LOC Mod.	Files Mod.	fixed directly	fixed later	never fixed	false	Total	Fix Time (day)
LibrelioDev	15120	4	2	1	212	5	0	16	0	15	31	54
Sonet	18294	14	0	0	708	4	0	0	12	20	32	–
SocializeSDK	65032	3	1	0	191	4	0	0	0	2	2	–
ChatSecureAndroid	35220	3	0	0	187	3	5	0	7	7	19	–
GwindowWhat	8785	4	0	0	68	4	0	0	0	0	0	–
Irccloud	21384	3	0	0	82	3	3	8	3	0	14	124
Cyclestreets	13523	3	0	0	98	3	0	0	0	0	0	–
Owncloud	17016	3	0	0	108	3	4	0	0	8	12	–
AndroidSettings	71226	5	0	0	117	4	0	0	0	0	0	–
AndroidCalendar	34090	4	0	0	113	4	0	0	0	0	0	–
MyExpenses	20914	2	0	0	34	2	0	0	0	0	0	–
Allplayers	5693	21	0	0	361	18	3	12	0	5	20	1
GRTransit	3316	4	1	0	206	5	0	2	2	0	4	14
Total	329613	73	4	1	2394	62	15	38	24	57	134	193

expanded the selected expression into a full statement, and then supplied it as the input to ASYNCHRONIZER.

After changing the input source code to pass preconditions, we applied ASYNCHRONIZER to these four cases and included them along with the other metrics shown in Tab. 3.2.

Accuracy: Column 5 shows the number of differences between manual and automated refactorings. The differences do not include other changes made by developers (e.g., adding new features). There is only one case in `LibrelioDev-Android` project where the two outputs differ. In this case, manual refactoring moves fewer statements into `onPostExecute` handler, but they don't affect the semantics, which means that the code behaves the same way in both cases.

Effort: In total, the refactoring modified 2394 lines of code in 62 files (see `LOC Mod.` and `Files Mod.` columns in Tab. 3.2). On average, each refactoring changes 31 lines of code. More important, many of these changes are non-trivial: programmers need to infer fields, method parameters, and return value, which statements can be moved into `onPostExecute`, as well as deal with special cases. In contrast, when using ASYNCHRONIZER, the programmer only has to initiate the transformation. ASYNCHRONIZER takes less than 10 seconds per refactoring.

Safety: Columns 8 to 12 show the 134 races that ASYNCHRONIZER detected automatically and we checked manually.

Notice that 38 races are not fixed immediately in the manual refactoring, but are fixed in a later version. The strategies to fix these races include adding synchronizations, moving the statements involved in races outside of `AsyncTask`, changing shared variables into local variables, or removing the shared variables. Interestingly, among these 38 races, 12 races in `Allplayers-Android` project are fixed incorrectly the first time: developers invoke `get` immediately after executing the `AsyncTask`. They applied a second patch to fix them correctly in a later version. In the four projects that fix races in a later version, developers spent 193 days in total to apply patches (`Fix Time` column). There are 57 false warnings. The reasons for the false warnings were discussed in Sec. 3.5.3.

The remaining 24 races still exist in the latest version. We reported all of them to developers. They fixed 3 races in `Irccloud-Android`, and they confirmed 9 races in `ChatSecureAndroid` and `GRTransit`. The developers of `Sonet` do not think the 12 races lead to bugs. In this case, the pair of racing accesses are in two event handlers which developers confirmed can not happen in parallel (code examples are on [52]). In practice, developers can avoid checking such races by customizing synthetic call graphs based on their domain knowledge about which event handlers may happen in parallel.

Our result shows 62 (columns `fixed later` + `never fixed`) out of 134 races are neither detected nor fixed when developers perform manual refactoring. Even when they are fixed in a later version, the timespan is long. Thus, ASYNCHRONIZER is safer than manual refactoring.

Value: Table 3.3 shows results where we used ASYNCHRONIZER to refactor long-running operations from main thread into `AsyncTask`. We used a corpus of 19 projects, where in total we applied ASYNCHRONIZER to 123 places in 72 files. 121 cases satisfied the preconditions. Similar to the previous experiment, for the two cases that failed the preconditions, we manually modified the code to satisfy the preconditions. We also check the races reported by ASYNCHRONIZER (column 5). Notice that the races we show in Tab. 3.3 do not include false warnings (there are 72 false warnings in total).

We created patches which include the transformations and fixes for races, and submitted

Table 3.3: Results of applying ASYNCHRONIZER to 19 projects that have potential responsiveness issues.

Project Name	SLOC	Passed	Failed	Races	Replied?	# Accepted	Files Mod.
Connectbot	33326	24	0	70	No	0	5
Cgeo	66389	11	0	0	Yes	11	6
KeePassDroid	28588	1	2	0	Yes	3	3
Vudroid	2408	1	0	13	Yes	0	1
VLC	36852	13	0	16	Yes	0	8
Rpicheck	6859	13	0	0	Yes	13	5
Adblockplus	11970	2	0	0	Yes	0	2
Alfresco	70471	7	0	0	No	0	7
HockeySDK	7052	4	0	0	Yes	4	4
FBReaderJ	58718	10	0	8	No	0	5
Catlog	6035	1	0	0	No	0	1
Glimmr	9570	3	0	0	No	0	4
K-9 Mail	78679	7	0	38	Yes	7	5
Open311	6642	5	0	0	No	0	4
Spika	19823	3	0	0	No	0	1
Eoecn	11760	3	0	0	No	0	2
Andlytics	44441	2	0	0	Yes	2	2
Liberario	8171	9	0	0	Yes	0	6
Allplayers	5693	2	0	0	No	0	2
Total	513447	121	2	145	-	40	72

the patches to the open-source developers. Column 6 shows whether developers replied to our patches. We got 10 replies. Columns 7 shows the number of patches developers accepted and merged. In total, developers from 6 projects have accepted 40 refactorings. The developers of `Vudroid`, `VLC` and `Liberario` do not think the operations encapsulated into `AsyncTask` significantly affect UI responsiveness. For example, `Vudroid` developers said "*`ZoomRoll` class instance is a singleton for application and hence your patch will change only the first time load delay. I have never observed considerable time delays on Activity start*". The developers of `Adblockplus` think the `AsyncTask` we introduced can lead to data races on external files, so they do not accept our refactorings. However, detecting data races on external resources is beyond the ability of ASYNCHRONIZER. This result shows the importance of having domain knowledge, but also shows that our refactoring approach can produce useful results accepted by developers.

CHAPTER 4

Converting Shared-Memory into Distributed-Style Communication for Android Apps

4.1 Introduction

In addition to `AsyncTask`, Android framework also provides other two major async constructs: `IntentService` and `AsyncTaskLoader`. However, as discussed in Chapter 1, `AsyncTask` can lead to memory leaks, lost results, and wasted energy if improperly used. In general, `AsyncTask` is designed for encapsulating short-running tasks while the other two are good choices for long-running tasks.

To learn how async constructs are used in Android apps, understand how developers retrofit asynchrony, and learn about barriers encountered by developers, we first conducted a formative study by analyzing a corpus of 611 most popular open-source Android apps, comprising 6.47M SLOC. We then surveyed 10 expert Android developers to put the study results in a broader context. The study answers the following questions:

RQ1: *How do Android developers use asynchronous programming?* Mapping the landscape of usage of async constructs in Android is useful for researchers, library designers, and is educational for developers. We found that 161 (32%) of the studied apps use at least one asynchronous programming, resulting in 1893 instances. Out of these, `AsyncTask` is the most widely used.

RQ2: *How do Android developers retrofit asynchrony into existing apps?* Must asynchrony be designed into a program, or can it be retrofitted later? What are the most common transformations to retrofit asynchrony? Answering this question is important for software evolution researchers, as well as tool builders. We found widespread use of refactorings, both from sequential code to async, and from basic async to enhanced async. We found the following code evolution scenario: developers first convert sequential code to `AsyncTask`, and those that continue to evolve the code for better use of asynchrony refactor it further into enhanced constructs.

RQ3: *How do expert developers interpret the disparity in usage of async constructs?* Answering this question can provide educational value for developers. We found that experts think `AsyncTask` is being overused at the expense of other enhanced async constructs, and many inexperienced Android developers do not know its problem. They also suggest `AsyncTask` should only be considered for short-running tasks (i.e., less than a second). They suggest that the current guides and examples of `IntentService` are not enough, and point out the need for refactoring tools to help unexperienced developers use and learn about the enhanced async constructs and the different style of communicating with the GUI.

Inspired by the results of our formative study, we designed, developed, and evaluated AsyncDroid, an automated refactoring tool that transforms shared-memory into distributed-style communication in the context of Android async programming. AsyncDroid refactors `AsyncTask` into `IntentService`. We developed AsyncDroid as an Eclipse plugin, thus it offers all the convenience of a modern refactoring tool: it enables the user to preview and undo changes and it preserves formatting and comments. To use it the programmer only needs to select an `AsyncTask` instance, then AsyncDroid verifies that the transformation is safe, and rewrites the code if the preconditions are met. However, if a precondition fails, it warns the programmer and provides useful information that helps the programmer fix the problem.

4.2 Background on Android IntentService and AsyncTaskLoader

We introduced the background of Android programming and `AsyncTask` construct in Sec. 3.2. Though `AsyncTask` is an easy-to-use construct, it is ideally used for short tasks. Otherwise, the three problems introduced in Chapter 1 (memory leaks, lost results, wasted energy) can occur. In this section, we introduce the other two major async constructs which are safer but more complicated: `IntentService` and `AsyncTaskLoader`.

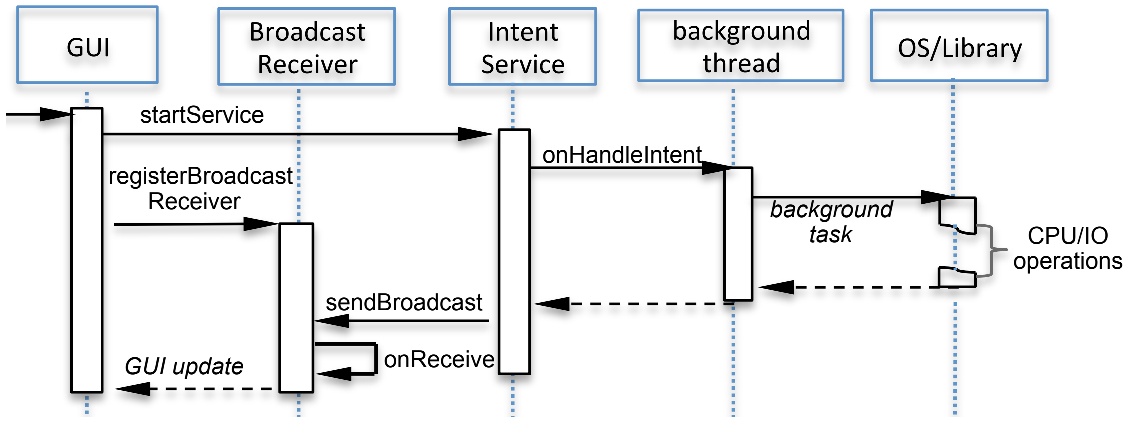


Figure 4.1: Where is `IntentService` code executing?

4.2.1 IntentService in Android

`IntentService` belongs to the *Service* component, but its encapsulated operations are executed in a background thread instead of UI thread. Figure 4.1 shows the flow of `IntentServ-ice`. Service is started when `startService` is invoked. Then `onHandleIntent` method is executed in a background thread. Unlike `AsyncTask`, `IntentService` uses a distributed-style programming to communicate with UI thread. To get the task result, the GUI that starts the service should register a broadcast receiver. After the task is finished, `IntentService` sends its task result via `sendBroadcast` method. Once the registered receiver receives (i.e., the GUI) this broadcast, its `onReceive` method will be executed on UI thread, so it can get the task result and update GUI.

Figure 4.2(a) shows a real-world example from *owncloud-android* app, which is an Android client for a cloud server. It uses `AsyncTask` to execute the background task. `LogActivity` starts a `LoadingTask` at line 11. The task reads some files in `doInBackground` (lines 32, 33) and returns a string (line 36). `onPostExecute` takes this string and uses it to update a text view (line 22). Notice that `LoadingTask` is declared as a non-static inner class, so it holds a reference to `LogActivity`.

Figure 4.2(b) shows an equivalent implementation of `LogActivity` using `IntentService`. The background task from Fig. 4.2(a) is now put into `onHandleIntent` method (line 32). The result is wrapped by an `Intent` object, which is marshalled and sent via broadcast (lines

```java
1  public class LogActivity extends Activity {
2
3
4    private String mLogPath=FileUtils.getLogPath(),
5    protected void onCreate() {
6      ...
7      LoadingTask task = new LoadingTask(view);
8
9
10
11     task.execute();
12   }
13   class LoadingTask extends AsyncTask {
14     private TextView textView;
15     private Exception exception;
16     LoadingTask(TextView view){textView = view;}
17     public void onPostExecute(String result) {
18
19
20
21       if (exception == null) {
22         textView.setText(result);
23       } else Log.i(exception.getMessage());
24     }
25
26
27
28     public String doInBackground(String[] args){
29
30
31       try {
32         File file = new File(mLogPath);
33         ...
34
35
36         return text;
37       } catch (Exception e) {
38         exception = e;
39
40
41         return null;
42 }}}}
```

(a) Using `AsyncTask`

```java
1  public class LogActivity extends Activity {
2    public static final FILTER = "LogActivity_receiver";
3    private LoadingReceiver receiver;
4    private String mLogPath = FileUtils.getLogPath();
5    protected void onCreate() {
6      ...
7      receiver = new LoadingReceiver(view);
8      registerReceiver(receiver, new IntentFilter(FILTER));
9      Intent intent= new Intent(this,LoadingService.class);
10     intent.putExtra("mLogPath", mLogPath);
11     startService(intent);
12   }
13   class LoadingReceiver extends BroadcastReceiver {
14     private TextView textView;
15     private Exception exception;
16     LoadingReceiver(TextView view){ textView=view; }
17     public void onReceive(Context ctx, Intent intent){
18       String text = intent.getStringExtra("RV");
19       Exception exception = (Exception)intent
20            .getSerializableExtra("exception");
21       if (exception == null) {
22         textView.setText(text);
23       } else Log.i(exception.getMessage());
24 }}}
25 public class LoadingService extends IntentService {
26   private String mLogPath;
27   private Exception exception;
28   public void onHandleIntent(Intent intent) {
29     mLogPath = intent.getStringExtra("mLogPath");
30     Intent result = new Intent(LogActivity.FILTER);
31     try {
32       File file = new File(mLogPath);
33       ...
34       result.putExtra("exception", exception);
35       result.putExtra("RV", text);
36       sendBroadcast(result);
37     } catch (Exception e) {
38       exception = e;
39       result.putExtra("exception", exception);
40       result.putExtra("RV", null);
41       sendBroadcast(result);
42 }}}
```

(b) Using `IntentService`

*`onCreate` method starts a background task to read a log file, and the result is shown in a `TextView`. (a) and (b) shows two semantic-equivalent implementations using `AsyncTask` and `IntentService`, respectively.

Figure 4.2: Asynchronous code from *Owncloud* app.

35, 36). `Intent` is analogous to a hash map whose key is a string and value is a serializable object (i.e., implements `java.io.Serializable` or `android.os.Parcelable`). It is the only medium through which different components (e.g., *service* and *broadcast receiver*) can exchange data. Finally, `onReceive` unwraps the result and updates the text view (line 22). Notice that to register a receiver, the developer should provide a filter (represented by a string) to specify which broadcast the receiver can receive. For example, line 2 defines a filter "FILTER". Lines 8 and 30 use it to register receiver and send broadcast.

Since `IntentService` is not affected by the destruction of the GUI objects that started it, it does not suffer from the problems introduced in Chapter 1. Thus, it can be safely used for both short or long tasks.

4.2.2 AsyncTaskLoader

`AsyncTaskLoader` is built on top of `AsyncTask`, and it provides similar handlers as `AsyncT-ask`. Unlike `AsyncTask`, `AsyncTaskLoader` is lifecycle aware: Android system binds/unbinds the background task with GUI according to GUI's lifecycle. Thus, it can also solve the problems we mentioned in Chapter 1. However, `AsyncTaskLoader` is introduced after Android 3.0, and it only supports two GUI components: *activity* and *fragment*.

4.3 Formative Study of Android Asynchrony

We want to assess the state of practice async programming in open-source Android apps. To obtain a deep understanding of asynchronous programming practices in Android, we answer three research questions. We now answer each of these questions, by first presenting the methodology and corpus, and then the results.

RQ1: How do Android developers use asynchronous programming?

To answer this question, we studied *all* async constructs provided by the standard Android libraries: `AsyncTask`, `IntentService`, `AsyncTaskLoader` and also the legacy-style Java `Thread`.

Corpus-1. To collect representative Android apps, we chose Github [50]. To distinguish Android apps in Github, we first searched all apps whose `README` file contains the "Android" keyword. Because we want to analyze recently updated apps, we filtered for apps which have been modified at least once since July 2014. Then, we sorted all these apps based on their

Table 4.1: Usage of async constructs in the Corpus-1

	# Instances	# App	App%
`AsyncTask`	938	97	19%
`Thread`	655	110	22%
`IntentService`	182	30	6%
`AsyncTaskLoader`	118	14	3%

star count and gathered the top 500 apps. To make sure that these apps have Android projects, we check whether every app has at least one *"AndroidManifest.xml"* file, which every Android project must contain in its root directory. After all filters, our code corpus has 500 Android apps, comprising 4.69M non-blank, non-comment SLOC (as reported by SLOCCount [39]).

Methodology. We built a tool, AsyncAnalyzer to automatically analyze the usage of async constructs in our corpus. AsyncAnalyzer uses Eclipse API for syntactic analysis. It builds an abstract syntax tree (AST) for each source file and traverses ASTs to gather the usage statistics for the four async constructs. Doing the analysis at the level of ASTs and not at the textual level, improves the accuracy in several ways.

First, it is immune to noise generated by text that matches names of the async constructs (e.g., import statements, comments, variable names, etc.) but does not represent an instance of an async construct. Second, it correctly accounts for the ways in which developers instantiate async constructs: via anonymous inner classes (e.g., `myAsyncTask = new AsyncTask(...)`) and via class subtyping (e.g., `class MyService extends IntentService`.

Results. Table 4.1 tabulates the usage statistics of async constructs. The three columns show the total number of construct instances, the total number of apps with instances of the construct, and the percentage of apps with instances of the construct.

As we see from the table, `AsyncTask` is the most popular async construct in our corpus, based on the total number of instances. However, if we count the total number of apps that use at least one of these constructs, then the legacy-style `Thread` is the most popular, despite the fact that Android already provides three special async constructs. `AsyncTaskLoader` and `IntentService` are not as popular as the other two.

RQ2: How do Android developers retrofit asynchrony into existing apps?

Next we analyze how developers introduce async constructs into their apps. We want to determine whether developers introduced async constructs when (i) implementing new features (i.e., asynchrony was added to a new program element when writing code from scratch), (ii) refactoring from existing sequential code, (iii) refactoring from another existing async construct.

In order to be able to detect transitions between basic and enhanced async constructs, we need to find projects where developers are aware that the enhanced constructs exist. Thus, we use 2 different corpora to study `IntentService` and `AsyncTaskLoader`, respectively:

Corpus-2. We collected 93 random open-source Android apps from Github, comprising 1.54M SLOC, which use both `AsyncTask` and `IntentService` constructs in their latest snapshot.

Corpus-3. We collected 18 random open-source Android apps, comprising 0.24M SLOC, which use both `AsyncTask` and `AsyncTaskLoader` constructs in their latest snapshot.

Methodology. In order to identify transitions, we study not only the latest version of the code, but also the first version where developers introduce async constructs. To do this, we automatically searched the commit history of our corpora through Gitective API [51], identified the commits that add import statements to `AsyncTask`, `AsyncTaskLoader`, or `IntentService`. After automatically finding commits that introduce these async constructs, we manually inspected the versions before and after such commits in order to understand how these async constructs are introduced.

Results. Tables 4.2 and 4.3 show how `AsyncTask`, `IntentService`, and `AsyncTaskLoader` are introduced.

The results show that in many cases developers refactor sequential code to `AsyncTask`. This observation confirms our previous findings [21]. However, the refactorings for `IntentSe-rvice` and `AsyncTaskLoader` mostly come from other async constructs. This shows the following code evolution scenario: developers first convert sequential code to `AsyncTask`, and those that continue to evolve the code for better use of asynchrony refactor it further into `IntentService` or `AsyncTaskLoader`.

RQ3: How do expert developers interpret the disparity in usage of async constructs?

To shed light into this question, we conducted a survey with expert Android developers.

Methodology. To find expert developers, we used StackOverflow [61], which is the pioneering Q&A website for programming. In StackOverflow, users are sorted by their points that they received from their answers for questions which are associated with some tags. We contacted the top 10 users for the “android-async” tag and these 10 people are the ones who

Table 4.2: How developers introduce `AsyncTask` and `IntentService` in the Corpus-2

Type	# Instances
Newly added `AsyncTask`	277
Refactor sequential code to `AsyncTask`	103
Refactor `Thread` to `AsyncTask`	18
Newly added `IntentService`	205
Refactor sequential code to `IntentService`	13
Refactor `Thread` to `IntentService`	18
Refactor `AsyncTask` to `IntentService`	9
Refactor `AsyncTaskLoader` to `IntentService`	5

Table 4.3: How developers introduce `AsyncTask` and `AsyncTaskLoader` in the Corpus-3

Type	# Instances
Newly added `AsyncTask`	73
Refactor sequential code to `AsyncTask`	24
Refactor `Thread` to `AsyncTask`	2
Newly added `AsyncTaskLoader`	15
Refactor sequential code to `AsyncTaskLoader`	3
Refactor `AsyncTask` to `AsyncTaskLoader`	10
Refactor `Thread` to `AsyncTaskLoader`	3

answered the questions related to Android async programming most. On average, each of them answered 2095 questions on StackOverflow. We got replies from 5 of them, including the author of a popular Android programming book [62].

Results. We asked three questions and summarized the experts' answers below:

Q1) Why are there still lots of legacy-style `Thread` uses even though Android provides three dedicated async constructs?

First, `Thread` construct has been around since the beginning and many Android developers formerly developed Java apps. Developers are very familiar with `Thread` and they do not have time to learn something new, thus they continue using it. Second, `Thread` is suitable for other scenarios, such as parallelism and scheduled tasks.

Q2) Why are there many more `AsyncTask` uses than the other two enhanced constructs even though `AsyncTask` may lead to memory leaks, lost task results, and wasted energy?

They all agree that `AsyncTask` "*is being overused*" at the expense of the other two enhanced constructs. As a main reason, they invoke a historical account "*`AsyncTask` was advertised to the developers a lot in Android documentation*" and it "*got a lot of good press early on*". On the other hand, they thought "*many developers coming from desktop do not realize the async nature of the Android memory management*". They also mention that `AsyncTaskLoader` has been around only for a short time and is harder to implement, and Google has not provided production-level examples of code that use `IntentService` and `AsyncTaskLoader`.

As a guidance on Android async programming, they suggest that `AsyncTask` or `Thread` should only be considered for short tasks (i.e., less than one second). For the work that will take more than a second, developers should use `IntentService`.

Q3) Do you think that developers can benefit from a refactoring tool from `AsyncTask` to `IntentService`?

They concluded that the technical challenges make the automation really hard: "*it would be a very difficult task, so the end solution may appear very complicated*", "*that would be quite a challenge to do automatically*". One also said that "*it may help beginner, but senior developers still like using their preferred way of writing async*" and another said "*it would have to be very compelling for users to take their existing code and change it - especially to something they do not already understand*".

Discussion: The answers from experts show that `AsyncTask` is easier to use and has better guides and examples than the other two constructs. Also, many developers do not understand Android memory management thus do not know the dark side of `AsyncTask`. This observation explains why `AsyncTask` is the most popular async construct, and why most of the time developers refactor sequential code to `AsyncTask` but seldom refactor `AsyncTask` to other constructs. On the other hand, automated refactoring between these constructs is challenging, but can provide helpful coding examples for developers.

4.4 Refactoring AsyncTask to IntentService

Based on our findings from Sec. 4.3, developers tend to choose `AsyncTask` for Android async programming. However, `AsyncTask` is not fit for long-running tasks, where developers should use `IntentService` or `AsyncTaskLoader`. Inspired by these findings, we propose AsyncDroid, an automated refactoring tool that transforms shared-memory into distributed-style communication in the context of Android async programming. AsyncDroid refactors existing `AsyncTask` into `IntentService`. It helps developers migrate the inappropriately used `AsyncTasks` to `IntentServices`. Additionally, by looking at transformations performed (e.g., using Eclipse's preview refactoring feature), developers can educate themselves on how to transform between `AsyncTasks` to `IntentServices`.

4.4.1 Refactoring Challenges

There are three main challenges that make it hard to execute the refactoring quick and flawlessly by hand. First, the developer needs to determine which objects should be transferred into `IntentService` and `BroadcastReceiver`, and how to transfer them. Second, the developer should establish channels to enable communication between `IntentService` and GUI. Third, the developer must register the receiver properly in order to receive the computation result from the established channel.

Transfer Objects from/to `IntentService`. As shown in Sec. 4.2, the non-local objects required by `AsyncTask` are passed as method arguments or can be directly accessed as fields from the outer class. However, the objects that flow into and escape from `IntentService` have to be wrapped and sent via an `Intent` object. Similar to distributed-style programming, objects transferred in this way are required to be serializable [63].

For example, at line 32 in Fig. 4.2(a), field `mLogPath` flows into `doInBackground` method. Thus, it should be transferred to `IntentService` during refactoring (line 10 in Fig. 4.2(b)). Determining objects transfer requires a nontrivial inter-procedural analysis of the code: the developer must trace (i) the call graph to figure out which objects flow into `IntentService` and `BroadcastReceiver`, and (ii) type hierarchy to check if the objects can be serialized.

Establish Channels for Communication. `AsyncTask` provides four handlers that enable developers to interact with GUI. Background task and handlers exchange data by accessing the shared memory. However, `IntentService` sends broadcast to `BroadcastReceiver` to communicate with GUI. Thus, the developer needs to rewire the channels of communication. To achieve this, one must split `AsyncTask` into `IntentService` and `BroadcastReceiver`. Splitting an `AsyncTask` includes moving the related fields and methods into `IntentService` and `BroadcastReceiver`, which requires to trace the call graph. Additionally, the developer has to write extra code for sending broadcast, which makes the refactoring more tedious.

Where to Register the Receiver. In order to receive the computation result from the channels, the GUI needs to register a `BroadcastReceiver`. A naive approach is to register at the original call site where the `AsyncTask` is created. For example, in Fig. 4.2(a), the task is created in `onCreate` at line 7. While using `IntentService` in Fig. 4.2(b), the receiver is registered at the same place in `onCreate` (line 8). This approach only works when the original call site is already in a lifecycle event handler, such as `onCreate` method. Lifecycle events are guaranteed to be triggered by OS during GUI recreation, so the receiver can be registered automatically.

However, if the call site is not in a lifecycle event handler, the receiver cannot be registered unless the event is triggered again after GUI recreation. For example, the call site can be in the `onClick` listener of a button, so the receiver can only be registered when the user clicks the button. If the GUI containing the button is recreated while the background task is running, the recreated GUI cannot receive the broadcast unless the button is clicked again. Thus, the developer also needs to infer where to register `BroadcastReceiver` during the refactoring. This is a non-trivial insight for developers, because online documents only show basic Android asynchronous programming scenarios, where this is not a concern.

4.4.2 The Canonical Form of AsyncTask Code and Refactoring Preconditions

A key insight in designing refactorings is that there is a canonical form for input code [64, 65]. This canonical form adheres to the preconditions of the refactoring, so that the result of the transformation is indeed correct. This means that if the input code is not in canonical form, it is necessary to transform into canonical form before performing the refactoring. AsyncDroid has several preconditions, which together dictate the canonical form which the source code must adhere to for a successful transformation:

P1: *All variables that flow into or escape from `doInBackground` are or can be marked as serializable.* This is required because such variables need to be transferred to `IntentService` and `BroadcastReceiver` via marshalling/unmarshalling.

P2: *All the methods invoked in `doInBackground` should also be accessible by `IntentService`.* Because most AsyncTasks are declared as non-static inner classes, they can call methods from the outer class. Since IntentService is not an inner class, it needs to be able to call the same set of methods from the outer class, so their visibility needs to be appropriate.

P3: *The refactored task is directly extended from `AsyncTask` and is not subclassed.* This precondition prevents the refactoring from breaking the inheritance relation and affecting other `AsyncTasks` that are not refactored in the type hierarchy.

P4: *An `AsyncTask` instance is only used when invoking `AsyncTask.execute`.* For example, if a task instance is used as a method argument or return value, AsyncDroid halts the refactoring to avoid changing the design contract of the method. On the other hand, `AsyncTask` defines some methods that are not supported by `IntentService` (e.g., `AsyncTask.cancel`). If these methods are invoked on the task instance, AsyncDroid halts the refactoring.

Figure 4.2(a) is a valid example of a target program that meets all these preconditions, thus it can be refactored by AsyncDroid. We will see in Sec. 4.5 how many real-world programs readily meet these preconditions.

4.4.3 The Refactoring Algorithm

AsyncDroid takes the following steps to refactor an `AsycnTask` to `IntentService`.

Analyzing Transferred Objects. Since `doInBackground` and `onHandleIntent` are semantic-equivalent methods that enclose background task, AsyncDroid needs to analyze `doInbackground` to determine which objects should be transferred.

We define the *Target Class*, *Incoming Variables* and *Outgoing Variables* for `IntentService` as following:

Definition 1 (Target Class ($\mathcal{TC}$)) *The top-level or static inner class that creates and starts the* `AsyncTask`*.*

Definition 2 (Incoming Variables ($\mathcal{IV}$)) *The set of non-local variables flow into* `doInB-ackground`*, which have to be transferred to* `IntentService`*, is:*
$F_{TC} \cup F_{task} \cup Args_{task} \cup LV_{TCM}$ *where:*

- F_{TC} *is the set of* $\mathcal{TC}$*'s fields when the* `AsyncTask` *is a non-static inner class of* $\mathcal{TC}$
- F_{task} *is the set of* `AsyncTask`*'s fields that are initialized in its constructors or* `onPreEx-ecute` *handler*
- $Args_{task}$ *are the arguments of* `AsyncTask.execute` *method*
- LV_{TCM} *is the set of* **final** *local variables declared in the* $\mathcal{TC}$*'s method where the task is created, when the* `AsyncTask` *is an anonymous inner class of* $\mathcal{TC}$

Notice that collecting $\mathcal{IV}$ requires inter-procedural analysis since `doInBackground` may invoke other methods defined in $\mathcal{TC}$ or the `AsyncTask`.

Definition 3 (Outgoing Variables ($\mathcal{OV}$)) *The set of variables that escape from* `doInBac-kground` *and need to be transferred from* `IntentService` *to* `BroadcastReceiver`*, is:*
$F^M_{TC} \cup F^M_{task} \cup RV$ *where:*

- F^M_{TC} *is the set of* $\mathcal{TC}$*'s fields that are modified in* `doInBackground` *method*
- F^M_{task} *is the set of* `AsyncTask`*'s fields that are modified in* `doInBackground` *method and used in* `onPostExecute`
- RV *is the return value of* `doInBackground` *method*

An example of $\mathcal{OV}$ is the return value `text` and a field `exception` in Fig. 4.2(a) (lines 36, 38). F^M_{TC} and F^M_{task} are $\mathcal{OV}$ because `IntentService` and GUI holds and operates on different copies of objects due to (de)serialization. They should be written back in `BroadcastReceiver` otherwise the modifications are lost. Note that there is no way to write back modified incoming LV_{TCM} or $Args_{task}$ due to the communication mechanism of `IntentService`. However, we never find a case in practice where these two types of $\mathcal{IV}$ are modified. A reasonable explanation is that modifying them in a background thread can introduce data races.

As mentioned in Sec. 4.4.1, the objects in the $\mathcal{IV}$ and $\mathcal{OV}$ set are required to be serializable. AsyncDroid traverses the type hierarchy and checks the serializability of an variable type based on its definition [66]: a type is serializable if it implements `java.io.Serializable` and all of its fields' types are serializable, unless the field is ***transient***. AsyncDroid also refactors a type to implement `java.io.Serializable` if its fields conform to the definition but it has not implemented yet. Note that when refactoring a type to be serializable, AsyncDroid also checks the serializability of all its subtypes to guarantee the transformation is safe, and it only refactors the type defined in source code, not libraries. If any of the $\mathcal{IV}$ and $\mathcal{OV}$ are not serializable or cannot be refactored, the refactoring fails precondition **P1**.

Generating and Starting `IntentService`. First, for the target `AsyncTask`, AsyncDroid creates a corresponding `IntentService` class. The method body of `onHandleIntent` is moved from `doInBackground`. Unlike `AsyncTask`, `IntentService` class cannot be a non-static inner class. Thus, AsyncDroid creates it as an independent class. AsyncDroid creates a field for each variables in $\mathcal{IV}$ and $\mathcal{OV}$. AsyncDroid adds statements to unwrap objects in $\mathcal{IV}$ from `Intent` at the beginning of `onHandleIntent` (line 29 in Fig. 4.2(b)) . At every exit of `doInBackground`, AsyncDroid creates an `Intent` to carry $\mathcal{OV}$, and sends it via broadcast (lines 34 to 36 and 39 to 41 in Fig. 4.2(b)).

Second, since `doInBackground` can invoke methods defined in `AsyncTask` or $\mathcal{TC}$'s methods, AsyncDroid copies such methods into `IntentService` class. Note that AsyncDroid moves such methods instead of copying if `doInBackground` is the only caller. However, if any of such methods are in library code and cannot be copied, the refactoring fails **P2**.

Finally, AsyncDroid rewrites the call sites of `AsyncTask.execute` into `startService`.

This includes creating the `Intent` object to wrap the $\mathcal{IV}$ (lines 9-11 in Fig. 4.2(b)).

Notice that in Android, starting a service or sending a broadcast requests a `Context` object. AsyncDroid checks if (i) $\mathcal{TC}$ itself is a subclass of `Context` and (ii) $\mathcal{TC}$ contains a visible `Context` field or local variable, or a visible method that returns a `Context` object. AsyncDroid uses such `Context` if there is any, otherwise the refactoring stops.

Creating and Registering Receiver. `AsyncTask` communicates with GUI through handlers. However, to receive task result from `IntentService`, the developer needs to establish a channel by registering a `BroadcastReceiver` on GUI. AsyncDroid rewrites the target `AsyncTask` into `BroadcastReceiver` class. It keeps all the fields, constructors and handlers defined in `AsyncTask`, and rewrites `onPostExecute` handler into `BroadcastReceiver.onReceive` (line 17 in Fig. 4.2(b)). AsyncDroid also inserts statements at the beginning of `onReceive` to unwrap $\mathcal{OV}$ and writes them back to corresponding variables (lines 18, 19 in Fig. 4.2(b)).

As discussed in Sec. 4.4.1, to avoid losing task result during GUI destroying and recreation, the receiver should be registered in lifecycle event handlers. AsyncDroid declares the receiver as a field of $\mathcal{TC}$ (line 3 in Fig. 4.2(b)). It tries to move the receiver creation and registration into $\mathcal{TC}$'s lifecycle event handlers unless they are already there. The following example shows an `AsyncTask` that executes in a button's `onClick` listener. AsyncDroid register the receiver in `onCreate` lifecycle handler during refactoring instead of in the `onClick` listener:

```java
void onCreate() {
    button.setOnClickListener() {() -> { new MyAsyncTask(...).execute();}}
}
```

$$\Downarrow$$

```java
BroadcastReceiver receiver;
void onCreate() {
  receiver = new MyBroadcastReceiver(...);
  registerReceiver(receiver, ...);
  button.setOnClickListener() {() -> {startService(...);}}
}
```

Note that a $\mathcal{TC}$ can have multiple lifecycle event handlers. AsyncDroid registers the receiver in the handler that is invoked first by the system (e.g., `onCreate`). A receiver can be moved if and only if (i) the $\mathcal{TC}$ contains lifecycle event handlers, (ii) all variables transferred to the receiver are still visible to it after moving, and (iii) the variables used by receiver's constructor are not redefined in other lifecycle event handlers. Rule (ii) guarantees syntax correction while rule (iii) preserves the semantics of the refactoring. AsyncDroid raises a warning when a receiver cannot be moved.

A filter is required when registering receiver. The filter specifies which broadcast a receiver can receive. AsyncDroid concatenates $\mathcal{TC}$ class name and receiver name as filter name, and uses it to register receiver and send broadcast (lines 2, 8 and 30 in Fig. 4.2(b)).

Dealing with other `AsyncTask` handlers. In addition to `onPostExecute`, `AsyncTask` provides three other handlers. Note that the generated `BroadcastReceiver` keeps all these three handlers. For `onPreExecute`, AsyncDroid inserts an invocation to this handler before starting the service.

For `onProgressUpdate`, AsyncDroid first rewrites the call site of `publishProgress` into `sendBroadcast`, with the filter set to "ProgressUpdate":

```java
void doInBackground(...) { publishProgress(arg); }
```

$$\Downarrow$$

```java
void onHandleIntent(...) {
     intent.setAction("ProgressUpdate");
     intent.putExtra("taskProgress", arg);
     sendBroadcast(intent);
}
```

Then in `onReceive`, AsyncDroid adds a branch to intercept the "ProgressUpdate" broadcast, and invokes `onProgressUpdate`:

```java
void onProgressUpdate(...) {...}
void onReceive(Context context, Intent intent) {
    ... // code from onPostExecute
}
```

$$\Downarrow$$

```java
void onProgressUpdate(...) {...}
void onReceive(Context context, Intent intent) {
  if(intent.getAction().equals("ProgressUpdate")) { ...; onProgressUpdate(...); }
  else ... // code from onPostExecute
}
```

AsyncDroid ignores `onCancelled` handler, since Android does not support canceling `IntentService`. Tasks that invoke `AsyncTask.cancel` fails **P4**.

4.4.4 The Refactoring Implementation

We have implemented the refactoring as a plugin in the Eclipse IDE, on top of Eclipse JDT [41] and refactoring engine [42]. To use AsyncDroid, the developer selects the `doInBackground` method in the `AsyncTask` she wants to transform, and then chooses Convert To IntentService option from the refactoring menu.

Limitation: When searching for the call sites of `execute` for a task, AsyncDroid searches in the syntax block where the task is created and compares the task variables via JDT variable binding. AsyncDroid also checks if the task variable is assigned to only one task instance in the searched block (i.e., no alias on the task variable in the searched block). The refactoring fails **P4** if no call site is found. Although this heuristic approach can miss refactoring chances, such as `execute` is called in a different method, we only find one case in our study and experiment where this approach does not apply.

4.5 Evaluation

To empirically evaluate whether AsyncDroid is useful, we answer the following evaluation questions.

EQ1. Applicability: How applicable is the refactoring?

EQ2. Effort: How much programmer effort is saved by AsyncDroid when refactoring?

EQ3. Accuracy and Value: How accurate is AsyncDroid when performing a refactoring? Do developers think that the refactorings performed by AsyncDroid are useful?

Table 4.4: Nine popular Android projects from Github.

Project Name	SLOC	# AsyncTask	# IntentService
Owncloud	54918	6	0
Open311	6642	5	0
Prey-android-client	15361	8	1
SMSSync	16706	10	2
Opentripplanner	11766	7	0
UltimateAndroid	228154	10	0
AntennaPod	38430	24	0
WhatAndroid	23643	20	0
TextSecure	40819	18	0
Total	436439	102	3

4.5.1 Experimental Setup

To answer the above questions, we apply AsyncDroid on nine popular Github open-source Android projects. We selected projects that use `AsyncTask` predominantly.

Table 4.4 provides statistics about these projects. We report the size (in SLOC), the number of `AsyncTask` and `IntentService` instances that are used in each project.

For each project, we applied the AsyncDroid to every `AsyncTask`, except for the ones that have already been used in `Service` or retained `Fragment`. The lifecycle of such `AsyncTasks` is independent of GUI's lifecycle, so the they do not suffer from the problems described in Chapter 1.

We recorded several metrics for each refactoring. To measure the applicability, we counted how many instances met the refactoring preconditions and thus can be refactored. We also analyzed the reasons why the remaining instances cannot be refactored by AsyncDroid. To measure refactoring effort, we recorded the number of input and output variables ($\mathcal{IV}$, $\mathcal{OV}$), serialized types, moved/copied methods and moved receivers. We also counted the number of files and SLOC that are changed. To verify the accuracy and value, we manually examine the correctness of all the refactored code. We also sent 45 refactorings in 7 projects to developers and let them judge the correctness and usefulness.

Table 4.5: Applicability of applying AsyncDroid to `AsyncTask` in nine Android projects.

Project Name	Passed	Conditional Passed	Failed	P1	P2	P4
Owncloud	1	2	3	4	1	2
Open311	5	0	0	0	0	0
Prey-android-client	7	1	0	1	0	0
SMSSync	1	0	6	6	0	0
Opentripplanner	2	2	3	4	1	0
UltimateAndroid	5	1	4	2	2	2
AntennaPod	4	4	16	15	4	16
WhatAndroid	9	0	8	6	0	1
TextSecure	10	0	3	3	0	0
Total	44	10	43	41	7	21

$\mathcal{IV}$: incoming vars; $\mathcal{OV}$: outgoing vars; **P1**: all $\mathcal{IV}$ and $\mathcal{OV}$ can be serializable; **P2**: all methods invoked by `doInBackground` are accessible in `IntentService`; **P4**: `AsyncTask` is only used when invoking `AsyncTask.execute`.

Table 4.6: Effort of applying AsyncDroid to `AsyncTask` in nine Android projects.

Project Name	# $\mathcal{IV}$	# $\mathcal{OV}$	Moved Methods	Serialized Types	Moved & unmoved Receivers	Files Mod.	SLOC Mod.	# Task Objects
Owncloud	6	5	1	0	2/0	3	193	3
Open311	6	10	3	10	3/0	11	395	5
Prey-android-client	11	9	5	2	4/2	10	392	8
SMSSync	0	1	0	0	1/0	1	41	1
Opentripplanner	8	8	4	3	0/10	14	679	10
UltimateAndroid	6	6	0	0	4/0	7	310	8
AntennaPod	8	8	1	2	4/3	11	418	8
WhatAndroid	11	10	0	0	9/0	9	411	9
TextSecure	25	10	0	1	3/0	11	547	10
Total	81	67	14	18	30/15	77	3386	62

$\mathcal{IV}$: incoming vars; $\mathcal{OV}$: outgoing vars.

4.5.2 Results

Table 4.5 and 4.6 shows the result of applying AsyncDroid on the `AsyncTasks` in our corpus of 9 Android projects.

Applicability. Table 4.5 shows the applicability. We totally refactored 97 `AsyncTasks` in the nine projects. Columns 2 and 4 show the number of instances that pass and fail the refactoring preconditions. There are 44 `AsyncTasks` that pass the preconditions, while 43 fail the preconditions. This is not a limitation of AsyncDroid, but such cases can not be converted from shared-memory to distributed-style. Column 3 shows another 10 `AsyncTasks` that fail the preconditions. However, these instances can be refactored into canonical form with other well-known refactorings (such as demoting fields to local variables) so that AsyncDroid can refactor them. We show these instances in the "conditional pass" column.

We discovered two common transformations that convert code into canonical form. First, an unserializable object contained in $\mathcal{IV}$ but not in $\mathcal{OV}$, can be converted into a local variable in `doInBackground` as long as it is only used by `doInBackground`:

```java
UnserializableObject object = new UnserializableObject(...);
void doInBackground(...) { object.method(); }
```

$$\Downarrow$$

```java
void doInBackground(...) {
    UnserializableObject object = new UnserializableObject(...);
    object.method();
}
```

Second, an `AscynTask` that is executed on a `ThreadPool`, can be changed to execute on a plain thread since `IntentService` does not support `ThreadPools`:

```java
AsyncTask task = new AsyncTask() {...};
task.executeOnExecutor(...);
```

$$\Downarrow$$

```java
AsyncTask task = new AsyncTask() {...};
task.execute(...);
```

For refactorings that conditional-pass or fail the preconditions, we analyzed which preconditions they violate. Columns 5 to 7 shows the number of instances that fail **P1**, **P2** and **P4**. Note that one refactoring can violate multiple preconditions. The result shows most failed refactorings violate **P1**. The main reason is that the unserializable types in $\mathcal{IV}$ or $\mathcal{OV}$ are declared in third-party libraries (such as network or database). A source-to-source transformation tool like AsyncDroid cannot transform third-party binary code. A typical scenario is an `AsyncTask` access network or database in background thread, however, network or database APIs are usually in libraries.

Preconditions **P2** and **P4** are violated mainly due to methods `cancel` or `executeOnExecutor` that are invoked either in `doInBackground` (thus failing **P2**) or on the task instance (thus failing **P4**). Since these methods are specific to `AsyncTask` and `IntentService` does not support them, AsyncDroid cannot transform those cases. For **P3**, we only find one violation in *WhatAndroid*, so we do not show them in Table 4.5 due to lack of space.

In terms of applicability, AsyncDroid successfully refactored 45.3% `AscynTasks` directly in nine projects. There are 10.3% `AscynTasks` that can also be refactored after converting them to canonical form. This shows that AsyncDroid has a high level of applicability.

Effort. Table 4.5 shows the effort. We estimate the effort based on the 54 `AsyncTasks` that pass or conditional pass the preconditions. In the last column, we show the number of task instances that are created for the 54 `AsyncTasks`. 62 task instances are created in total, which means most `AsyncTasks` are used only at one place. This observation confirms our previous study [21]: developers tend to tightly bind an `AsyncTask` to only one GUI component.

Columns 2 and 3 show the number of $\mathcal{IV}$ and $\mathcal{OV}$ for each project. For the 54 `AsyncTasks`, there are 81 $\mathcal{IV}$ and 67 $\mathcal{OV}$. Detecting them needs inter-procedural analysis. Moreover, wrapping them into `Intent` object is also tedious.

Column 4 shows the methods that need to be moved or copied into `IntentService`. We find 14 methods that should be put into `IntentService`. Note that searching these methods also needs inter-procedural analysis. Column 5 shows the number of types that are refactored to be serializable. On average, each refactoring marks 0.33 types as serializable. However, checking serializability is tedious since it requires traversing the type hierarchy for each field.

Column 6 shows the number of `BroadcastReceivers` that are moved into lifecycle event handlers by AsyncDroid (left side of slash), and that have to be moved but AsyncDroid cannot move (right side of slash). Notice that for each task instance, AsyncDroid creates a corresponding receiver. Therefore, it creates 62 receivers in total: 30 are moved, 15 cannot be moved, and the remaining 18 are already in lifecycle event handlers. For the 15 unmoved receivers, GUI component can lose task result if GUI destroying and recreation occurs during task running, thus AsyncDroid raises a warning.

Columns 7 and 8 show the number of files and SLOC that are changed during the 54 refactorings. On average, each refactoring changes 1.43 files and 63 SLOC. AsyncDroid helps developers change several SLOC, and such changes are non-trivial. Thus, we conclude that AsyncDroid can save developers' effort.

Accuracy and Value. By manually examining the 54 refactored instances applied by AsyncDroid, we determined that no compilation errors were introduced and the original semantics of `AsyncTask` are preserved.

We also submitted 45 refactorings in 7 projects to developers through Github pull request. Given the large size of changes in the patches that we submitted (on average a patch touching 10 files with 403 additions and 210 deletions), we expected that developers might not reply - as previous studies [19, 46] show that open-source developers are more likely to respond to small patches.

Despite this, we still received replies from 4 projects in which 15 patches were accepted. *WhatAndroid* [67] developers accepted the 9 refactorings we submitted by saying: "*this is an interesting set of changes, AsyncTask can definitely be a pain to deal with*" and "*these tasks are a good fit for migration to IntentServices and I will migrate over to IntentServices*". *AntennaPod* [68] developers said their `AsyncTasks` are short: "*most of our tasks read or write data from the database and should finish well under 100ms*". They think `AsyncTasks` work fine for their short-running tasks while `IntentService` makes the code more verbose. This shows that AsyncDroid can produce accurate and valuable results.

CHAPTER 5

A Platform for Practical Impact

5.1 Introduction

Although static analysis and refactoring tools help developers find and fix bugs, these tools can be difficult to smoothly integrate with each other and into the developer workflow, particularly when scaling to large codebases [69]. Therefore, Google developes ShipShape, a program analysis platform aimed at building a data-driven ecosystem around program analysis. Google envisions ShipShape to become a widely-used platform. Any app developer that wants to check code quality, for example before submitting an app to the app store, would run ShipShape on her code base.

ShipShape [17] is a static program analysis platform developed by Google. It allows customized analyzers to plug in through a common interface. ShipShape platform relies on docker, which is a lightweight virtual machine. Docker allows one to package an application with all of its dependencies into a standardized docker image for software development. A docker image can wrap up an application in a complete filesystem that contains everything it needs to run, including code, runtime libraries, system tools, etc [18].

Figure 5.1 shows the high-level architecture of ShipShape. ShipShape is packaged in a docker image (i.e., the ShipShape service in Fig. 5.1). When the image is running, ShipShape service starts up, listens to certain ports and sends analysis requests that are sent to these ports (i.e., the dash arrows in Fig. 5.1). Customized analyzers (i.e., the analyzer service in Fig. 5.1) can send analysis results to the ports that ShipShape service is listening to (i.e., the solid arrows in Fig. 5.1). Note that customized analyzers are also deployed in docker images. A customized analyzer can send analysis results to ShipShape through the interfaces it provides. The results contain the location of the detected buggy code (i.e.,

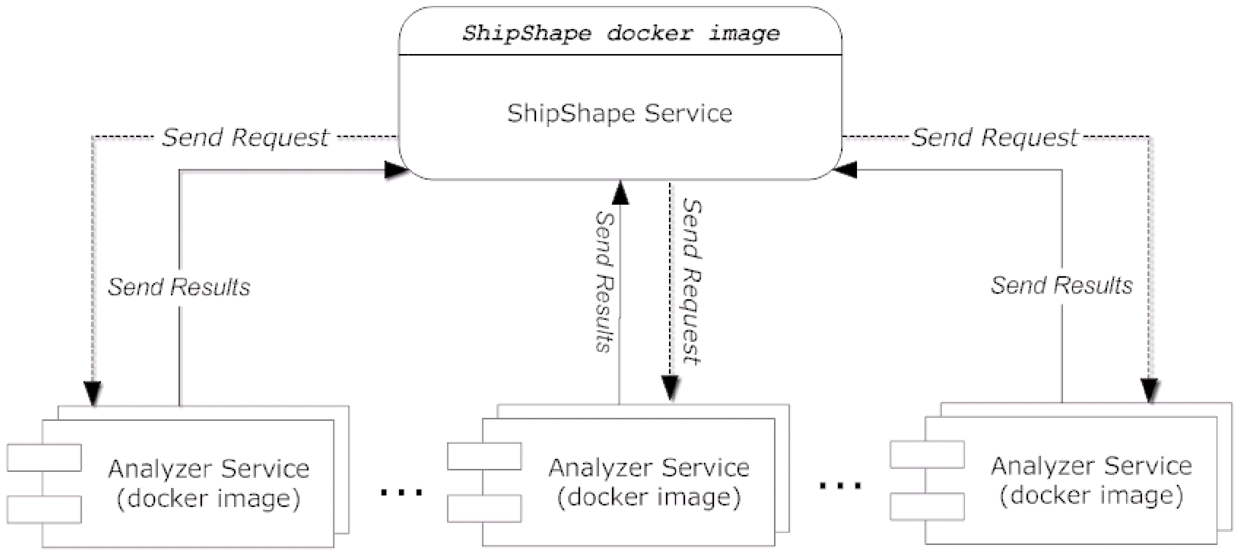


Figure 5.1: High-level architecture of ShipShape.

file name, line and column numbers) and the fixes for the buggy code. ShipShape can print the results in JSON [70] format. Once ShipShape receives a command, it invokes the corresponding analyzer and gets the results. ShipShape can be run either as a command-line interface, or as a Jenkins plugin [71].

In order to increase the practical impact of this dissertation, we integrated CTADetector and AsyncDroid with ShipShape. We expect that by contributing new async analyzers to ShipShape, millions of app developers would benefit by being able to execute our analysis and transformations on their code. Note that Asynchronizer is not integrated because Asynchronizer is a semi-automated refactoring tool: developers need to select the code that they want to extract into `AsyncTask`. However, Asynchronizer cannot accurately infer which piece of code should be extracted. Thus, Asynchronizer does not match ShipShape's philosophy which only supports fully automatic analyzers.

In this chapter, we will present the technical details of integrating with ShipShape. Integrating our toolsets with ShipShape includes three major steps: (i) implementing Eclipse headless plugins; (2) implementing ShipShape interface; (3) building docker images that contain our toolset.

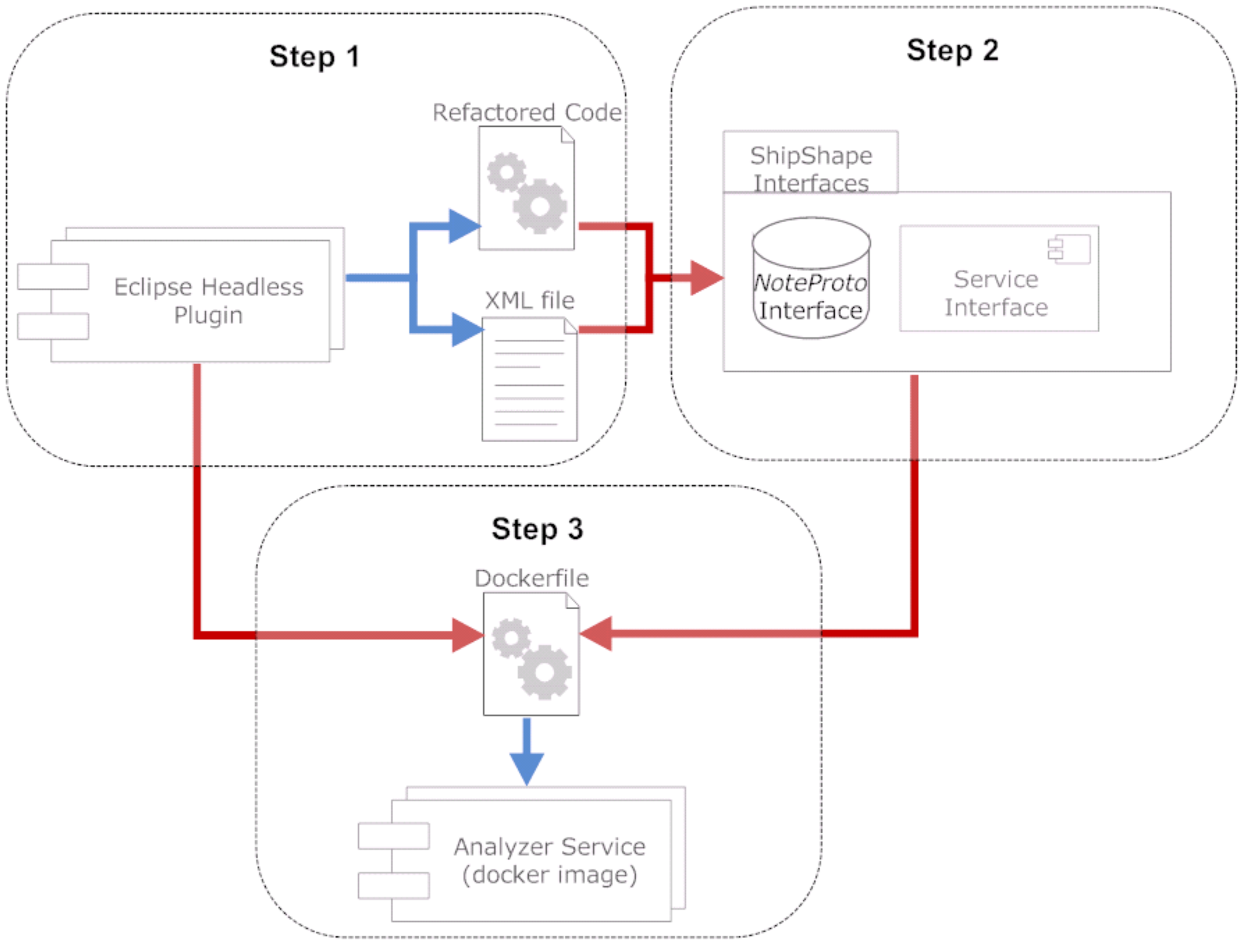


Figure 5.2: Steps of integrating Eclipse plugin with SHIPSHAPE.

5.2 Technical Details of Integrating with ShipShape

We implemented our toolset as Eclipse plugins. In order to integrate with SHIPSHAPE, we should first be able to run the Eclipse plugins in a docker image. Figure 5.2 shows the three major steps of integrating Eclipse plugin with SHIPSHAPE. The blue arrows show the outputs of each step, while the red arrows show the inputs of each step.

Since docker images cannot run Eclipse with GUI, it requires running the Eclipse plugins in headless mode [72] from command line. Thus, we first modified our implementation to support Eclipse headless mode. In this modified implementation, instead of showing results through Eclipse GUI widgets (e.g. Eclipse refactoring wizards), the tools save them as a XML file. CTADETECTOR saves the locations of the detected incorrect idiom instances, while

AsyncDroid saves the locations of all the `AsyncTask`. The refactoring results are used as fixes for ShipShape. The tools save the refactored code (i.e., the entire file that contains the refactored code) in temporary files. These files will be used to generate ShipShape fixes so their locations are also saved in the XML file. Thus, the output of running CTADetector and AsyncDroid in Eclipse headless mode is a XML file that records the locations and a set of temporary files that contain the refactored code. This step is shown as "Step 1" in Fig. 5.2, whose outputs are the XML file and the refactored code.

The second step is to implement ShipShape interfaces to receive analysis request and send analysis results to ShipShape. To receive analysis request, we need to define an analyzer service, which includes creating analyzer, initiating analyzer category and stage, and registering the service to certain port so ShipShape can listen to it. To write analysis results, ShipShape provides a *NoteProto* interface. This interface defines several fields to specify the locations of the files that contain the buggy code, the line and column numbers of the buggy code. Our implementation parses the XML file generated by our Eclipse headless plugins to retrieve this information. For fixes, *NoteProto* provides fields to select a piece of code by its start and end position, and developers can set a replacement to replace this piece of code with correct code. In our implementation, we use a third party library *java-diff-utils* [73] to compute the diffs between the original files and the refactored files (i.e., the temporary files saved by our headless Eclipse plugin in the first step). Such diffs contain the insertions, deletions and replacements in term of line numbers. For example, in Fig. 4.2, the computed diffs between Fig. 4.2(a) and Fig. 4.2(b) includes inserting "*private LoadingLogReceiver receiver;*" at line 3, and replacing "*task.execute();*" with "*this.startService(intent);*" at line 11, etc. We then use these diffs to set the fixes for *NoteProto*. "Step 2" in Fig. 5.2 represents this step, in which we need to implement service interface and *NoteProto* interface using the outputs of the first step.

Finally, we have to create a docker image that contains our toolset. This docker image will be used as plugin of ShipShape. Since docker images are virtual machines, we can install all the dependencies that required by our toolset in a docker image. This includes JVM, the Eclipse that contains our headless plugins and the executable Java archive files (i.e., the *jar* package) implemented in the second step. To build a docker image, we have to define a *Dockerfile* [74]. In *Dockerfile*, we can add instructions to build up the virtual machine we need. In our implementation, we use *apt-get* to install JVM and *wget* to download the Eclipse, and execute the *jar* package implemented in the second step to start our analyzer service. After that, we can build docker image from *Dockerfile* [17, 74]. "Step 3" in Fig. 5.2 represents this step, in which we build a docker image that contains Eclipse headless plugin and the implementation of ShipShape interfaces. We also submitted our docker images to DockerHub [75] so they are publicly available.

Our implementation of integrating with ShipShape is available on our tool websites [40, 76]. To build and run customized analyzers on ShipShape, one can follow the instructions on ShipShape website [17]. Our tool websites [40, 76] also contain the instructions for running our toolset on ShipShape.

CHAPTER 6

Related Work

We briefly present the work that is closest to the dissertation. We organize the related work into: (i) empirical studies for concurrent programming, refactoring and API usage; (ii) detection of data races and atomicity violations; (iii) pattern-based program analysis; (iv) analysis and Testing for mobile apps; (v) safety analysis of event-driven applications; (vi) refactoring for concurrency, parallelism and asynchrony and (vii) compiling shared-memory programs to distributed-memory programs.

6.1 Empirical Study for Concurrent Programs, Refactoring and API Usage

Many works focus on studying concurrency bugs. Lu et al. [34] categorized concurrency bug types by analyzing a large number of bug reports from open-source repositories. They listed one of the six types of atomicity violations that we classified in Tab. 2.1. In a followup work [77] they also described bugs that manifest as performance slowdowns in concurrent programs. Schaefer et al. [78] showed several examples of how sequential refactorings can break concurrent programs. While Lu et al. [34] focused on the causes of the concurrency bugs, Fonseca et al. [79] presented a study of concurrency bugs in MySQL, in which they categorized and analyzed the effects (e.g., program crash, assertion violation) of the concurrent bugs. Pankratius et al. [80] studied the current state-of-the-practice in shared-memory, multicore programming, and suggested areas and engineering principles for future research. Weslley et al. [81] studied how developers are retrofitting applications to become more concurrent and summarized the usage of concurrent programming constructs in Java by analyzing more than 2000 projects. Li et al. [82] studied and categorized bug characteristics in modern software. Their result shows concurrency and performance related bugs can have a severe impact on software. Our previous work [83] tested and studied the behaviors of distributed applications under abnormal deploying environments.

Several researchers studied the software evolution and refactoring [84, 85, 19, 20, 86, 45, 46, 87, 88, 89, 90]. Kavaler et al. [84] studied how programmers ask questions about Android APIs on StackOverflow. Kim et al. [87] studied the benefits of refactoring in industrial code bases. Bavota et al. [90] investigated to what extent refactoring activities induce faults.

There are also several empirical studies [91, 92, 93, 94] on the usage of libraries or programming language constructs. Dyer et al. [95] analyzed 31k open-source Java projects to find uses of new Java language features over time. Buse et al. [96] proposed an automatic technique for synthesizing API usage examples and conducted a study on the generated examples. Semih et al. [8] studied how developers use Microsoft's Parallel Libraries. One of their findings is that some library constructs are error-prone. Also, Dig et al. [97] studied the evolution of Java concurrent applications and cataloged the changes that programmers made in response to concurrency. Another work [98] on automated refactoring to introduce concurrent library constructs shows that manual refactorings from `HashMap` to `ConcurrentHashMap` are error-prone. This dissertation focuses on the study of how programmers use, misuse and underuse of Java concurrent constructs.

6.2 Data Race and Atomicity Checking

Several researchers proposed dynamic [24, 29, 30, 9, 99, 100] or static techniques [25, 26, 33, 101, 31] to check data races or atomicity violations in concurrent programs. Some approaches [24, 9] require programmers to provide test drivers, but constructing test drivers for large applications is time consuming. Others [30] require programmers to write annotations, but industry programmers are reluctant to write annotations. Static techniques like [25] can report a large number of false warnings which may overwhelm programmers. A recent work [26] uses techniques such as better modeling of concurrent threads and synchronization constructs, as well as bubbling up the races from the library to application code level, which significantly reduce the number of false warnings. Dynamic techniques, such as Atomizer [102] and CTrigger [103], infer the unserializable interleavings automatically, and expose bugs by executing those interleavings. However, the inferred interleavings depend on the dynamic traces they use. Line-Up [104] is a dynamic linearizability checker that enumerates schedules without knowing atomic regions. Model checkers like JPF [99] and Chess [100] can expose concurrent bugs, including the bugs related to concurrent collections, but they might not be scalable to large applications.

Among these techniques, COLT [9] is a recent dynamic tool that checks the atomicity of composed operations from Java concurrent collections. COLT invokes the non-commutative operations before and after the tested operations, and uses linearizability as the test oracle. COLT found 41 problematic atomicity violations in 25 open-source projects. For the same projects used in COLT's evaluation, CTADETECTOR found 178 violation instances, from which we reported 85, and 55 of them are confirmed to be new bugs. Notice that even though we tried our best to make a fair comparison (e.g., we contacted the COLT authors), the comparison is hard because the COLT authors do not report how many of their bugs are confirmed by developers, neither the exact version number of their subject programs.

The advantage of COLT is that it reports fewer false positives than static analysis tools, since it uses the accurate dynamic information from the runtime execution instead of a static approximation. However, COLT needs test cases to execute the program and its results depend on the quality of the test cases. If the test cases do not cover the buggy idioms, COLT will miss such bugs (this could explain why we found more buggy instances, even confirmed by developers). In practice, CTADETECTOR and COLT can be used in tandem. For example, we can use COLT to check whether the reported instances by CTADETECTOR are real bugs.

Compared with other more general techniques, CTADETECTOR focuses on the atomicity violations of composed Java concurrent collection operations. Our approach does not require programmer to write tests or annotations. In addition, we also detect performance bugs that involve over-synchronization, and we tie the automated detection with interactive correction.

6.3 Pattern-based Analysis

Pattern inference and identification is also a widely used approach to improve software quality. `AVIO` [105] and `Falcon` [106] analyze the access patterns of variables to detect or locate concurrency bugs. `FindBugs` [107] detects bugs by statically matching the bug patterns to programs. The current version of `FindBugs` only considers one single variation of `put-if-absent` idiom out of our nine idioms. Yu et al. [108] exploited interleaving idioms to test concurrent programs. Uddin et al. [109] inferred temporal API usage patterns that can be used to improve the API design and usage. Wendehals and Orso [110] proposed a dynamic technique to recognize design patterns in the programs.

The success of the previous technique in finding buggy idioms shows that despite the fact that the buggy idioms are not particularly deep, today's state-of-the-art systems are still riffle with such bugs. Thus, custom pattern-based analyses, like our current work on patterns of concurrent collection usage, can be quite effective.

6.4 Analysis and Testing for Mobile Apps

Liu et al. [12] empirically studied performance bug patterns in Android apps, and concluded that executing long-running operations in main thread is the main culprit. They also proposed an approach to detect such operations statically. Berardinelli et al. [111] introduced a framework for modeling and analyzing the performance of context-aware mobile software systems. Arijo et al. [112] proposed a model-based approach to analyze the performance of mobile apps, in which they represented state changes by graph transformation. Muccini et al. [113] analyzed the challenges in testing mobile apps, including performance and memory testing. Lillack et al. [114] proposed an approach to track load-time configuration for Android apps, which can help with tuning performance of Android apps. Yan et al. [115] proposed a test generation approach to detect memory leaks for Android apps. Yang et al. [13] tested the responsiveness of Android apps by adding a long delay after each heavy API call. Choi et al. [116] used machine learning to learn a model for smartphone apps and generated test inputs from the model. Jensen et al. [117] proposed a test generation approach to find event sequences that reach a given target line in smartphone apps. Concolic testing [118] and random testing [119, 120] is also applied to smartphone apps. However, our work on ASYNCHRONIZER is complementary to testing: we enable developers to use `AsyncTask` refactoring to eliminate the performance issues that are detected in testing.

6.5 Safety Analysis of Event-driven Applications

Using static analysis, Sai et al. [58] formulated a solution based on call graph reachability to detect GUI accesses from outside the main thread, whereas Zheng et al. [121] targeted data races due to asynchronous calls for Ajax applications. Recent work on dynamic race detectors for event-driven applications [122, 123, 124, 125] proposed a causality model for JavaScript and Android which they used to infer happens-before relationships between events. Model-checking based techniques have also been proposed for event-driven or GUI applications [126, 127, 128]. In future work, we propose to investigate how the above techniques of modeling event relationships can be integrated with ASYNCHRONIZER.

6.6 Refactoring for Concurrency, Parallelism and Asynchrony

The refactoring community has been recently pushing refactoring technology beyond its classic realm (i.e. in improving software design) into improving non-functional qualities such as performance through parallelism and concurrency. Schafer et al. [129] proposed a refactoring for replacing Java built-in locks with more flexible locks. Wloka et al. [130] presented a refactoring for replacing global state with thread local state. Schafer et al. [131] examined whether classic refactorings can be safely applied to concurrent programs. Dig et al. implemented several implemented several concurrency-related refactorings to improve *throughput*, *scalability* and *asynchrony* [98, 132, 133, 134, 54, 21, 135, 136, 137].

6.7 Compiling Shared-memory to Distributed-memory

Several compiler techniques [138, 139, 140, 141, 142] have attempted to translate shared-memory program to distributed-memory program. However, these techniques target high performance distributed computing. In our work of AsyncDroid, we presented a refactoring from a shared-memory construct to a distributed-style construct in the context of Android asynchrony.

CHAPTER 7

Conclusion and Future Work

Concurrent constructs are some of the key features provided by modern programming languages and libraries. Although concurrent constructs make concurrent programming easier, this dissertation shows that they can be still misused and underused. However, interactive program transformation tools can mitigate these problems.

Some programmers erroneously think that just by using thread-safe concurrent collections their code is thread-safe. Our study of 28 projects reveals nine common CHECK-THEN-ACT idioms that can result in atomicity violations. We found that the distribution of correct and misused idioms is not the same, which means that some idioms are more error-prone than others. This finding is important for library designers who can design more resilient APIs. It also provides educational value for developers who use concurrent collections. Using this corpus and our tool, CTADetector, we found 282 buggy instances. The developers examined 90 of them and confirmed 60 as new bugs, and applied our patch. While they confirmed 67% of the examined bugs, they claim that the remaining do not result in bugs. This reasoning requires deep understanding of the domain and concurrency model. In addition, we reported the misuses of `put-if-absent` idiom that we found in this paper to Doug Lea, the lead designer of `j.u.c.` package. This led them to improve some APIs in the latest JDK8 release (e.g., `compute-if-absent` in `ConcurrentHashMap`) in ways they expect will reduce the prevalence of errors and misuses.

Asynchronous execution of long-running tasks is crucial for the now ubiquitous mobile and wearable apps. Despite significant efforts to educate Android app developers on how to use async programming, developers can underuse or improperly use the primary construct, `AsyncTask`, which can lead to memory leaks, lost results, and wasted energy. Based on the study of 715 Android projects, we discovered that developers refactor their sync code into `AsyncTask`, and some go further into using safer (but more complex) async constructs. However, their manual refactoring introduces performance bugs or data races. In some cases, it took developers hundreds of days to find and fix these bugs. To help developers correctly use these concurrent constructs, we presented two refactoring tools. Asynchronizer automates refactoring sequential code to use `AsyncTask`. The refactoring is composed of two steps: a code transformation that moves user-selected code into `AsyncTask`, and a safety analysis that checks data races. We applied Asynchronizer on 32 Android apps. We found that the tool is widely applicable, it is accurate compared to manual code transformations, it saves programmers' effort, it is safer than manual refactoring, and open-source developers accepted 40 patches with refactorings created by Asynchronizer. AsyncDroid is a refactoring that converts from `AsyncTask` which uses a shared-memory style of communication to `IntentService` which uses a distributed style of communication. Our evaluation shows that AsyncDroid is applicable and accurate, and it saves effort. Developers accepted 15 refactorings generated by AsyncDroid, which shows that it is valuable. This result shows that the refactorings presented in this dissertation can be useful in practice.

We now present our plans for possible future work building upon our current contributions and results as described in Chapters 2, 3, 4 and 5:

Automatically Inferring Places for Asynchrony: Although Asynchronizer and AsyncDroid provide refactorings for introducing concurrent constructs in existing code, developers still need to decide the places where to apply the refactorings to get the benefit of asynchrony. Despite extensive programming documentation (e.g., Android Best Practices for Performance [143]) or tools that detect I/O blocking operations (e.g., StrictMode [144] for Android), programmers still lack knowledge about where to introduce concurrency in the program. Thus, we need approaches for inferring locations where concurrency refactorings should be applied.

A naive approach is to put every potential long-running operations into concurrent constructs. For example, developers can use Android StrictMode to detect network, database and disk access, and encapsulate all these accesses into asynchronous task. However, this will result in unnecessary asynchronous code, since not every such access is long-running. For example, a single insertion to a local database might not need to be put into a background thread. On the other hand, developers have to ensure against data-races when performing concurrency refactorings, which is tedious and time-consuming. Therefore, it is not worth to apply concurrency refactoring to every blocking operations.

We plan to use two approaches to address this problem. First, we plan to apply data mining techniques [145, 146, 147] to mine frequently used API patterns in existing asynchronous code. We can use the mined patterns to determine whether an API usage should be put into a concurrent construct. Second, we plan to combine testing with our refactoring techniques. Testing tools such as MonkeyRunner [148] and Traceview [149] can help locate performance bottlenecks for Android apps. We plan to use them for profiling and inferring where to introduce asynchrony and which concurrent construct to use.

Mining Refactoring Patterns for Other Concurrent Constructs: In this dissertation, we studied the use, misuse and underuse of concurrent collections and Android async constructs. We plan to study the usage and misusage of other concurrent constructs. For example, Java 8 adds a new concurrent construct `java.util.concurrent.CompletableFuture` [150] which provides a range of methods for composing or handling background tasks without blocking the caller thread. It gives developers standard techniques for executing continuations when a task completes, and various ways to combine tasks. Reactive programming [151] provides a way for composing asynchronous and event-based programs by using observable sequences. It extends the observer pattern [152] to support sequences of data and events, and adds operators that allow one to compose sequences together declaratively while abstracting away low-level threading, synchronization, thread-safety and concurrent data structures. The emergence of these new concurrent constructs gives us an opportunity to discover more bug patterns and concurrency-related refactorings. In this dissertation, we manually determined whether developers introduce a concurrency construct by writing code from scratch or by moving existing code into the construct. We plan to investigate automatic approaches that infer how a construct is introduced during program evolution. Such approaches can facilitate our refactoring mining process.

More Precise Data Race Checking for Mobile Apps: In this dissertation, we proposed a static data race detection approach for Android apps and combined it with Asynchronizer. However, this approach has a high false positive rates (i.e., greater than 50%). The high false positive rate prevents the practical use of Asynchronizer. We plan to combine Asynchronizer with some more precise dynamic data race detection techniques for Android apps [124, 125], and evaluate the new combinations. We also plan to increase the precision of our static race detection technique for Android apps. For example, we can build context–sensitive and flow-sensitive control flow graph for analysis. Then we can compare the static and dynamic race detections for Android apps. We hope by such ways, we can increase the practical impact of Asynchronizer.

Impact of our toolset on Software Development: We deployed our refactoring tools as analyzers for ShipShape. Our vision is that Shipshape can become a widely-used analysis platform in near future. Any developer that wants to check code quality, for example before committing a code change, would run ShipShape on her code base. We expect that by integrating our tools with ShipShape, millions of app developers would benefit by being able to execute our analysis and transformations on their code. We plan to monitor the development status and adoption of ShipShape, as well as how practitioners are using our tools.

REFERENCES

[1] "Task Parallel Llibrary (TPL)," August 2015, http://msdn.microsoft.com/en-us/library/dd460717.aspx.

[2] "Collections.Concurrent (CC)," August 2015, http://msdn.microsoft.com/en-us/library/dd997305.aspx/.

[3] "Threading Building Block (TBB)," August 2015, http://threadingbuildingblocks.org.

[4] "Java Concurrent Library," August 2015, http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html.

[5] "AsyncTask," August 2015, http://developer.android.com/reference/android/os/AsyncTask.html.

[6] "IintentService," August 2015, https://developer.android.com/training/run-background-service/create-service.html.

[7] "AsyncTaskLorder," August 2015, https://developer.android.com/training/run-background-service/create-service.html.

[8] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the 20th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '12, 2012, pp. 1–11.

[9] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav, "Testing atomicity of composed concurrent operations," in *Proceedings of the 2011 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11, 2011, pp. 51–64.

[10] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice.* Addison-Wesley Professional, 2005.

[11] W. Torres, G. Pinto, B. Fernandes, J. a. P. Oliveira, F. A. Ximenes, and F. Castor, "Are java programmers transitioning to multicore?: a large scale study of java floss," in *Proceedings of the compilation of the co-located workshops on SPLASH '11*, ser. SPLASH '11 Workshops, 2011, pp. 123–128.

[12] Y. Liu, C. Xu, and S. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proc. of the International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 1013–1024.

[13] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in Android applications," in *Proc. of the International Workshop on the Engineering of Mobile-Enabled Systems*, ser. MOBS '13, 2013, pp. 1–6.

[14] "The dark side of AsyncTask," August 2015, http://bon-app-etit.blogspot.com/2013/04/the-dark-side-of-asynctask.html.

[15] "Activitys, Threads and Memory Leaks," August 2015, http://www.androiddesignpatterns.com/2013/04/activitys-threads-memory-leaks.html.

[16] "Android's AsyncTask," August 2015, http://steveliles.github.io/android_s_asynctask.html.

[17] "ShipShave Repository," August 2015, https://github.com/google/shipshape.

[18] "Docker," August 2015, https://www.docker.com.

[19] Y. Lin and D. Dig, "Check-then-act misuse of Java concurrent collections," in *Proc. of the International Conference on Software Testing, Verification and Validation*, ser. ICST '13, 2013, pp. 164–173.

[20] Y. Lin and D. Dig, "A study and toolkit of check-then-act idioms of java concurrent collections," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 397–425, 2015.

[21] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for android applications through refactoring," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 341–352.

[22] Y. Lin and D. Dig, "Refactorings for android asynchronous programming," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, 2015.

[23] Y. Lin, S. Okur, and D. Dig, "Study and refactoring of android asynchronous programming," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, 2015.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[25] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, 2006, pp. 308–319.

[26] C. Radoi and D. Dig, "Practical static data race detection for java parallel loops," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA '13, 2013, pp. 178–190.

[27] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12, 2012, pp. 387–400.

[28] J. Huang, P. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proceedings of the 35th Annual ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014.

[29] D. Weeratunge, X. Zhang, and S. Jaganathan, "Accentuating the positive: atomicity inference and enforcement using correct executions," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11, 2011, pp. 19–34.

[30] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '08, 2008, pp. 135–145.

[31] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06, 2006, pp. 334–345.

[32] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, 1986.

[33] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03, 2003, pp. 338–349.

[34] 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*, ser. ASPLOS '08, 2008, pp. 329–339.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *Proceedings of the 7th European Conference on Object-Oriented Programming*, ser. ECOOP'93, 1993, pp. 406–431.

[36] G. Upadhyaya, S. Midkiff, and V. Pai, "Automatic atomic region identification in shared memory spmd programs," in *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10, 2010, pp. 652–670.

[37] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, 2011, pp. 389–400.

[38] "Concurrency-Interest," August 2015, http://cs.oswego.edu/pipermail/concurrency-interest/2006-May/002511.html.

[39] "SLOCCount," August 2015, http://www.dwheeler.com/sloccount/.

[40] "CTADerector," August 2015, http://refactoring.info/tools/CTADetector/.

[41] "Eclipse Java development tools (JDT)," August 2015, http://www.eclipse.org/jdt/.

[42] "Eclips Refactoring Engine," August 2015, https://www.eclipse.org/articles/Article-LTK/ltk.html.

[43] "Gartner." August 2015, http://www.gartner.com/newsroom/id/2153215.

[44] "Tablet Sales." August 2015, http://www.gartner.com/newsroom/id/2954317.

[45] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '12, 2012.

[46] S. Okur, D. Hartveld, D. Dig, and A. Deursen, "A study and toolkit for asynchronous programming in C#," in *Proc. of the International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 1117–1127.

[47] "JDK Swing Framework," August 2015, http://docs.oracle.com/javase/6/docs/technotes/guides/swing/.

[48] "The SWT Toolkit," August 2015, http://eclipse.org/swt/.

[49] "Android Processes and Threads," August 2015, http://developer.android.com/guide/components/processes-and-threads.html.

[50] "GitHub," August 2015, https://github.com.

[51] "GiTective," August 2015, https://github.com/kevinsawicki/gitective.

[52] "Asynchronizer home page," August 2015, http://refactoring.info/tools/asynchronizer.

[53] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[54] C. Radoi and D. Dig, "Practical static race detection for Java parallel loops," in *Proc. of the International Symposium on Software Testing and Analysis*, ser. ISSTA '13, 2013, pp. 178–190.

[55] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, DIKU, University of Copenhagen, 1994.

[56] "T.J. Watson Libraries for Analysis (WALA)," http://wala.sourceforge.net/wiki/index.php.

[57] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, ser. POPL '95, 1995, pp. 49–61.

[58] S. Zhang, H. Lü, and M. D. Ernst, "Finding errors in multithreaded gui applications," in *Proc. of the International Symposium on Software Testing and Analysis*, ser. ISSTA '12, 2012, pp. 243–253.

[59] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proc. of the ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '13, 2006, pp. 308–319.

[60] J.-D. Choi, A. Loginov, and V. Sarkar, "Static datarace analysis for multithreaded object-oriented programs," IBM Research Division, Thomas J. Watson Research Centre, Tech. Rep., 2001.

[61] "Stack Overflow." August 2015, http://stackoverflow.com.

[62] M. Murphy, *The Busy Coder's Guide to Android Development*. CommonsWare, 2009.

[63] "Android Intents and Intent Filters," August 2015, http://developer.android.com/guide/components/intents-filters.html.

[64] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[65] W. G. Griswold, "Program restructuring as an aid to software maintenance," Ph.D. dissertation, University of Washington, Seattle, WA, USA, 1992.

[66] "Java Serializability," August 2015, http://docs.oracle.com/javase/1.5.0/docs/guide/serialization/spec/serial-arch.html.

[67] "Whatandroid repository." August 2015, https://github.com/Gwindow/WhatAndroid.

[68] "Antennapod repository." August 2015, https://github.com/AntennaPod/AntennaPod.

[69] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *37th IEEE/ACM International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 598–608.

[70] "JSON," August 2015, https://en.wikipedia.org/wiki/JSON.

[71] "Jenkins Plugins," August 2015, https://wiki.jenkins-ci.org/display/JENKINS/Plugins.

[72] "Headless Eclipse," August 2015, http://wiki.ptidej.net/doku.php?id=headless_eclipse.

[73] "java-diff-utils," August 2015, https://code.google.com/p/java-diff-utils/.

[74] "Dockerfile reference," August 2015, https://docs.docker.com/reference/builder/.

[75] "DockerHub," August 2015, https://hub.docker.com/.

[76] "AsyncDroid Tool," August 2015, http://refactoring.info/tools/asyncdroid/.

[77] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 77–88.

[78] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, "Correct refactoring of concurrent java code," in *Proceedings of the 24th European Conference on Object-oriented Programming*, ser. ECOOP'10, 2010, pp. 225–249.

[79] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *The 40rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '10, 2010, pp. 221–230.

[80] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy, "Software engineering for multicore systems: an experience report," in *Proceedings of the 1st International Workshop on Multicore software engineering*, ser. IWMSE '08, 2008, pp. 53–60.

[81] W. Torres, G. Pinto, B. Fernandes, J. a. P. Oliveira, F. A. Ximenes, and F. Castor, "Are java programmers transitioning to multicore?: a large scale study of java floss," in *Proceedings of the compilation of the co-located workshops on SPLASH 2011*, ser. SPLASH '11 Workshops, 2011, pp. 123–128.

[82] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: An empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ser. ASID '06, 2006, pp. 25–33.

[83] Y. Lin, F. Ivancic, P. Joshi, G. Balakrishnan, M. Ganai, and A. Gupta, "Environment-sensitive performance tuning for distributed service orchestration," *Lecture Notes in Computer Science*, vol. 8969, pp. 209–223, 2015.

[84] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. T. Devanbu, and V. Filkov, "Using and asking: APIs used in the Android market and asked about in stackoverflow," in *SocInfo*, ser. Lecture Notes in Computer Science, vol. 8238, 2013, pp. 405–418.

[85] N. Cacho, E. Barbosa, T. Cesar, A. Garcia, T. Filipe, and E. Soares, "Trading robustness for maintainability: An empirical study of evolving C# programs," in *Proc. of the International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 584–595.

[86] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," in *Proc. of the International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 151–160.

[87] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, 2012, pp. 50:1–50:11.

[88] Y. Lin, S. Zhang, and J. Zhao, "Incremental call graph reanalysis for aspectj software," in *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ser. ICSM '09, 2009, pp. 306–315.

[89] S. Zhang, Z. Gu, Y. Lin, and J. Zhao, "Change impact analysis for aspectj programs," in *Proceedings of the 24th IEEE International Conference on Software Maintenance*, ser. ICSM '08, 2008, pp. 87–96.

[90] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? An empirical study," in *Proc. of the IEEE International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '12, 2012, pp. 104–113.

[91] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger, "How developers use the dynamic features of programming languages: The case of Smalltalk," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 23–32.

[92] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An empirical investigation into a large-scale java open source code repository," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10, 2010, pp. 11:1–11:10.

[93] S. Karus and H. Gall, "A study of language usage evolution in open source software," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 13–22.

[94] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and use of Java generics," *Empirical Softw. Engg.*, vol. 18, no. 6, pp. 1047–1089, Dec. 2013.

[95] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of AST nodes to study actual and potential usage of Java language features," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 779–790.

[96] R. P. L. Buse and W. Weimer, “Synthesizing API usage examples,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE ’12, 2012, pp. 782–792.

[97] D. Dig, J. Marrero, and M. Ernst, “How do programs become more concurrent: a story of program transformations,” in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, ser. IWMSE ’11, 2011, pp. 43–50.

[98] D. Dig, J. Marrero, and M. Ernst, “Refactoring sequential java code for concurrency via concurrent libraries,” in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE ’09, 2009, pp. 397–407.

[99] W. Visser, K. Havelund, G. Brat, and S. Park, “Model checking programs,” in *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, ser. ASE ’00, 2000, pp. 3–11.

[100] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, “Finding and reproducing heisenbugs in concurrent programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08, 2008, pp. 267–280.

[101] C. von Praun and T. Gross, “Static detection of atomicity violations in object-oriented programs,” in *Journal of Object Technology*, vol. 3, no. 2, 2004, pp. 1–12.

[102] C. Flanagan and S. Freund, “Atomizer: a dynamic atomicity checker for multithreaded programs,” in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’04, 2004, pp. 256–267.

[103] S. Park, S. Lu, and Y. Zhou, “Ctrigger: exposing atomicity violation bugs from their hiding places,” in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’09, 2009, pp. 25–36.

[104] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, “Line-up: a complete and automatic linearizability checker,” in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’10, 2010, pp. 330–340.

[105] S. Lu, J. Tucek, F. Qin, and Y. Zhou, “Avio: detecting atomicity violations via access interleaving invariants,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’06, 2006, pp. 37–48.

[106] S. Park, R. Vuduc, and M. J. Harrold, “Falcon: fault localization in concurrent programs,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE ’10, 2010, pp. 245–254.

[107] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[108] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: a coverage-driven testing tool for multithreaded programs," in *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12, 2012, pp. 485–502.

[109] G. Uddin, B. Dagenais, and M. P. Robillard, "Analyzing temporal API usage patterns," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11, 2011, pp. 456–459.

[110] L. Wendehals and A. Orso, "Recognizing behavioral patterns atruntime using finite automata," in *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, ser. WODA '06, 2006, pp. 33–40.

[111] L. Berardinelli, V. Cortellessa, and A. D. Marco, "Performance modeling and analysis of context-aware mobile software systems," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, ser. FASE 10, 2010, pp. 353–367.

[112] N. Arijo, R. Heckel, M. Tribastone, and S. Gilmore, "Modular performance modelling for mobile applications," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '11, 2011, pp. 329–334.

[113] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *Proceedings of the 7th International Workshop on Automation of Software Test*, ser. AST '12, 2012, pp. 29–35.

[114] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014, pp. 445–456.

[115] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in Android applications," in *Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering*, ser. ISSRE 13, 2013, pp. 411–420.

[116] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proc. of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13, 2013, pp. 623–640.

[117] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proc. of the International Symposium on Software Testing and Analysis*, ser. ISSTA '13, 2013, pp. 67–77.

[118] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, ser. FSE '12, 2012, pp. 1–11.

[119] C. Hu and I. Neamtiu, "Automating gui testing for Android applications," in *Proc. of the International Workshop on Automation of Software Test*, ser. AST '11, 2011, pp. 77–83.

[120] Y. Lin, X. Tang, Y. Chen, and J. Zhao, "A divergence-oriented approach to adaptive random testing of java programs," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09, 2009, pp. 221–232.

[121] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proc. of the International Conference on World Wide Web*, ser. WWW '11, 2011, pp. 805–814.

[122] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 251–262.

[123] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *Proc. of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13, 2013, pp. 151–166.

[124] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race detection for event-driven mobile applications," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 326–336.

[125] P. Maiya, A. Kanade, and R. Majumdar, "Race Detection for Android Applications," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 316–325.

[126] O. Tkachuk and M. Dwyer, "Environment generation for validating event-driven software using model checking," *IET Software*, vol. 4, no. 3, pp. 194–209, 2010.

[127] M. Dwyer, V. Carr, and L. Hines, "Model checking graphical user interfaces using abstractions," in *Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '97, 1997, pp. 244–261.

[128] J. Atlee and J. Gannon, "State-based model checking of event-driven system requirements," in *Proc. of the Conference on Software for Citical Systems*, ser. SIGSOFT '91, 1991, pp. 16–28.

[129] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Refactoring Java programs for flexible locking," in *Proc. of the International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 71–80.

[130] J. Wloka, M. Sridharan, and F. Tip, "Refactoring for reentrancy," in *Proc. of the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. FSE '09, 2009, pp. 173–182.

[131] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, "Correct refactoring of concurrent Java code," in *Proceedings of the 24th European Conference on Object-oriented Programming*, ser. ECOOP'10, 2010, pp. 225–249.

[132] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Transformation for class immutability," in *Proc. of the International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 61–70.

[133] A. Gyori, L. Franklin, D. Dig, and J. Lahoda, "Crossing the gap from imperative to functional programming through refactoring," in *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. FSE '13, 2013, pp. 543–553.

[134] D. Dig, "A refactoring approach to parallelism," *IEEE Software*, vol. 28, no. 1, pp. 17–22, 2011.

[135] D. Dig, J. Marrero, and M. Ernst, "How do programs become more concurrent? A story of program transformations." in *IWMSE'11: International Workshop on Multicore Software Engineering*, 2011, pp. 1–8.

[136] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "ReLooper: refactoring for loop parallelism in Java," in *Companion to the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, ser. OOPSLA Companion '09, 2009, pp. 793–794.

[137] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson, "Inferring method effect summaries for nested heap regions," in *Proc. of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09, 2009, pp. 421–432.

[138] P. Banerjee, J. Chandy, M. Gupta, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su, "The paradigm compiler for distributed-memory message passing multicomputers," *IEEE Computer*, vol. 28, pp. 37–47, 1994.

[139] J. Zhu, J. Hoeflinger, and D. Padua, "Compiling for a hybrid programming model using the lmad representation," in *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC '01, 2003, pp. 321–335.

[140] A. Basumallik and R. Eigenmann, "Towards automatic translation of openmp to mpi," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05, 2005, pp. 189–198.

[141] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '06, 2006, pp. 119–128.

[142] S.-J. Min and R. Eigenmann, "Optimizing irregular shared-memory applications for clusters," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08, 2008, pp. 256–265.

[143] "Google Play," August 2015, http://developer.android.com/training/best-performance.html.

[144] "Google Play," August 2015, http://developer.android.com/reference/android/os/StrictMode.html.

[145] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00, 2000, pp. 1–12.

[146] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Trans. on Knowl. and Data Eng.*, vol. 12, no. 3, pp. 372–390, May 2000.

[147] M. J. Zaki and C. jui Hsiao, "Charm: An efficient algorithm for closed itemset mining," in *Proc. of the SIAM International Conference on Data Mining*, ser. SDM '02, 2002, pp. 457–473.

[148] "MonkeyRunner," August 2015, http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[149] "Traceview," August 2015, http://developer.android.com/tools/debugging/debugging-tracing.html.

[150] "CompletableFuture," August 2015, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html.

[151] "ReactiveX," August 2015, http://reactivex.io.

[152] "ObserverPattern," August 2015, https://en.wikipedia.org/wiki/Observer_pattern.