Understanding Change Patterns of Concurrent Programming

Anonymous

Abstract—**Concurrent programming is pervasive in nowadays software development. But it is a well-known fact that concurrent programming is difficult and error-prone. Many methods and tools have been presented to alleviate the difficulty of developing concurrent programs. On the other hand, developers are making numerous changes to their projects constantly. We can understand concurrent programming better by examining these changes. This paper studies concurrent related change patterns from a large amount of commits from 7 popular and representative Java open-source projects. We first select concurrent related commits by a combination of textual analysis and machine learning. We then examine the selected changes to understand what are changed and why changes happen. We create a taxonomy of these change patterns. The taxonomy comprise changing lock type, changing lock instance, changing critical section, adding or removing volatile and thread-safe class replacement. We also have some interesting findings: (1) Some changes are contrary. Different developers may modify their code in an opposite direction. (2) Developers are using some code-checking tools like FindBugs to help them inspect their code, but sometimes these tools are not enough.**

I. Introduction

Concurrent programs are pervasive [1] in nowadays software development activities. Using concurrency rightly in programs can exploit the calculation ability better with the rapid development of multi-core system. However, concurrent programming is very hard [2], [3] because multiple threads, which access objects simultaneously or depend on each other, usually need complex synchronization and it is hard to debug concurrent programs for the uncertainty of thread interleaving which makes it difficult to reproduce the bug [4]. Developers often struggle with various of synchronization methods and potential concurrent bugs. There are much research about concurrent programming in the literature such as concurrency bug detection [5], [6], [7], concurrent programming model [8], [9] and some empirical work [10], [11].

Software evolution is another hot topic in software engineering research. Software projects evolves during years because of new functionalities, bugs, reorganization of code. A few of open source software platforms like github has been more and more popular in recent years and so do the projects hosted there [12], [13]. They hold a huge amount of software projects and their historic versions. Researchers have shown that software evolution history can provide much useful information for today's software development activities. Many recent studies focus on these topics of software evolution such as refactoring [14], [15], transformation patterns [16], [17].

Limitations Many empirical studies have been conducted in concurrent programming. Gu *et al.* [18] studied change history

of thread synchronization. They checked 250,000 revisions of four open-source projects to figure out how critical sections change. They also conducted case studies to understand how the changes solve performance problems and correctness problems. However, concurrency is not only reflected by critical sections, but also some other programming constructs like concurrent libiary usage, thread resource management. Study on critical sections are not enough to understand real-world thread synchronization. Pinto *et al.* [1] did a large-scale study on the usage of Javas concurrent programming constructs. They checked the usage of concurrent programming constructs in a large base of code. They had many findings such as more than 75% of the projects create threads of do some concurrency control and the adoption of concurrent package of Java is moderate (23% concurrent projects use it). But we all know that software is changing every day. Numerous commits are submitted per day. We want to know not only the usage of concurrent programming, but also how concurrent programs are changed.

Santos *et al.* [19] studied system specific, source code transformations. They found some sequences of changes are systematic. They defined them as transformation patterns. They identified some transformation patterns in real world software and studied their properties like they are system specific, or they were applied in a manual way. However, the change patterns of concurrent programs are less studied.

Benefits We study concurrent programs from a perspective of software evolution history and find many change patterns about concurrent programming. Understanding concurrent program change patterns is very beneficial.

1) Developers are facing many concurrent programming requirements now. But concurrent programming is notoriously error-prone [4] because of the complexity of thread synchronization and scheduling. Even experienced developers might also be confused sometimes. Our study gives developers some guidelines of writing concurrent programs such as using handy concurrent libraries to finish the job instead of rewriting them by yourself unless all the available concurrent libraries cannot satisfy your requirement and you are absolutely confident of your concurrent programming skills.

2) Automatic tools are needed to help developers inspect and revise concurrent programs with the help of history information. We can learn from existing change patterns [20]. There has already been some tools [21], [22], [23], but they usually look for concurrent bugs such as race detection, deadlock detection and atomicity violation without considering software evolution history.

```
1    commit a6092d771ec50cf9aa434c75455b842f3ac6c628
2    Threading / initialisation issues. Not all were valid. Make them volatile
         anyway so FindBugs doesn't complain.
```

This is a commit message of Mark Thomas. He is a member of the Apache Tomcat Project Management Committee, and senior software engineer at the Covalent division of SpringSource. He has submitted more than 10,000 commits to Tomcat. He said that he made the changes anyway to eliminate FindBugs' warning. This indicates that a experienced programmer also feels bothered with these problems.

Both project specific and project independent transformation patterns exist in real-world software projects. So we need some concurrent code refactoring tools to give advice of what code need change and perform the transformations automatically. It is a chance for IDE manufacturer to make the IDE more intelligent in inspecting and modifying the code. Developers will benefit a lot if such kind of automatic tools can actually help them with automating their development and maintenance.

Challenges However, this work has to face several challenges:

1) The scale of open source software is increasing explosively as a result of some open source code platforms have become more and more popular. The change history of the open source software is also vast. Our interest is concurrent related commit, but they are hidden in the massive commit history. It requires much time and effort to identify whether a commit is concurrent related or not if doing it manually. We would like to adopt some automatic methods. Simple keyword matching algorithm will not work well because some commits just add or remove functionalities rather than modify original code.

2) The changes of code usually have complex relationship with the context not only in the file where change happens but also other files. Some change patterns have implicit dependency on the existing code. This raises a challenge to identify real change patterns which can be applied to other context correctly.

Contributions Our main contributions are:

- We identify and classify change patterns into ? types in concurrent programs from 102,028 commits of 7 open-source projects. The most common types of change patterns are thread-safe class replacement, changing critical sections.
- We find some interesting findings: Some changes are contrary. Different developers may modify their code in an opposite direction. Developers are using some code-checking tools like findbugs to help them inspect their code, but sometimes these tools are not enough.
- We affirm that our change patterns can be applied to appropriate contexts from real world open-source projects.
- We give some inspirations to concurrent program or library developers and analysis tool developers. Automated tools can be improved to help developers with program transformations.

The rest of paper is organized as follows: Section 2 presents the methodology of our study. Section 3 presents our result and

TABLE I
PROJECTS INFORMATION (LOC AND #FILES ARE BOTH OF JAVA FILES)

Project	LOC	#Files	#Commits
Hadoop	1202764	7701	14930
Tomcat	301173	2192	17731
Cassandra	387980	2143	21982
Lucene-solr	918398	6310	26152
Netty	218131	2054	7759
Flink	414264	4068	9771
Guava	251205	1672	3850
Mahout	109584	1215	3703

discussion. Section 4 presents related work. Section 5 presents future work and Section 6 concludes.

II. METHODOLOGY

This section presents data set of our study, research questions and tool support.

A. Data set

We investigate 8 Java open-source projects from Github including Hadoop[1], Tomcat[2], Cassandra[3], Lucene-solr[4], Netty[5], Flink[6] and Mahout[7] as shown in Table 1. Some projects are mirrored in Github. They are all popular, large-scale, active, representative Java open-source projects and cover different areas like distributed computing, web server, database, information retrieval, I/O and machine learning. The Hadoop project develops open-source software for reliable, scalable, distributed computing and has become one of the most famous Java open-source software for many years. Tomcat is the most popular implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. Cassandra [24] is a database system which can Manage massive amounts of data, fast, without losing sleep. Lucene-solr is two projects together in one respository in Github. Lucene is a search engine library and solr is a search engine server which uses lucene. Netty is an event-driven asynchronous network application framework. Flink is an open source stream processing framework with powerful stream- and batch-processing capabilities. Mahout is a machine learning project. Table I shows the lines of code in Java, the number of Java files and the number of commits of each project. All the projects are checked out for our study in December 2016.

B. Research questions

In order to understand the evolution of concurrent code and guide the future development better, we proposed 4 research questions:

RQ1. What are change patterns in concurrent programming?

---

[1]http://hadoop.apache.org/
[2]http://tomcat.apache.org/
[3]http://cassandra.apache.org/
[4]http://lucene.apache.org/
[5]http://netty.io/
[6]https://flink.apache.org/
[7]http://mahout.apache.org/

Researchers found that many code changes are similar [25]. Similar changes of code can be extracted into change patterns [26]. Some change patterns are project-specific while others are global, which can be considered as knowledge. Developers made numerous commits to the project repository during software's whole lifetime. There are a great many change patterns in software history. On the other hand, concurrent programming is very popular in today's Java development with the rapid developments of multi-core techniques which help exploit the power of concurrent programming. It is very meaningful to understand change patterns in concurrent programming. What are these change patterns and how many types of these change patterns are there?

RQ2. How frequent do concurrent related code modification appear in different kinds of Java open-source projects?

Java programming language provides convenient built-in concurrent libraries and users can also invoke third-party libraries like Apache Commons and Guava, which are both very famous libraries providing reusable conponents. Although developers can use their own concurrent related classes or third-party libraries, they are always using the facilities provided by Java standard libraries in most cases except they are facing special and rigour requirement. Previous researches [1], [27], [28] investigated the usage of concurrent libraries. We want to know how frequent do concurrent related code change occur in software projects. What are the differences of frequency in different kinds of software projects?

RQ3. What is the trend of concurrent programming construct usage statistically?

Java programming language offers many handy facilities for building concurrent programs. For example, there are language level constructs like `synchronized` and `volatile`. They are keywords of Java. There are also API level constructs like notify method of an object and some concurrent related convenient classes such as the `java.util.concurrent` package. There are always more than one ways to finish a task in Java and the preferences of developers evolves fast. We are interested in the trend of some common concurrent related constructs and the reasons hidden behind the phenomenon.

RQ4. Can these change patterns be applied to real world projects?

In order to better demonstrate these change patterns can really help developers understand concurrent programming practice and apply the practice to their projects, we are going to find the appropriate context in open-source projects and pull requests of applying the change patterns. We are insterested to see that how developers will accept our code change suggestions.

C. Tool support

We have developed a tool to collect and analyze data. The tool has the following parts.

1) Collecting commits: All the projects of our study are under git which is one of the most popular version control systems in the world. Some projects of the study used svn or some other version control systems before because they have long histories, but they all support git [29] now. We employ JGit, a lightweight, pure Java library implementation of git, to retrieve all the commit logs in projects' histories. A typical commit log contains commit id which is a 20-character-long string uniquely identifying a commit, author, date and message. Once we get a commit id, we use `git show` command to show the log message and textual `diff`. The `diff` result contains one or more change files which contain one or more change hunks.

2) Classification: There are many commits which are not concurrent related in the commits which we have collected. We need to select concurrent related commits. Tian *et al.* gave a successful example of identifing bug fixing patches using machine learning [30]. We use machine learning to train and predict whether a commit is concurrent related. We adopt both text analysis and code analysis to extract features. A commit log uses natural language to present what was changed and why the change was made in most cases. We treat each commit log as a bag of words then match the words to a set of concurrent keywords which we have defined as the Java concurrent keywords like `synchronized`, `volatile` and names of common classes or interfaces in Java libraries which are related to concurrency. We also do a code analysis based on the `diff` result. 12 features are extracted for each commit, which is shown in Table II. The first column shows the feature names and the second column shows the explanations.

We use the SVM [31] algorithm to train and classify commits as concurrent-related or not. SVM is a supervised classification algorithm which needs both positive and negative labeled data for training. In our tool, we use an implementation of SVM, LIBSVM [32]. We manually label some data as a training data set first then train a model. The trained classifier selects 135 positive instances from all the commits which we have collected.

D. Threats to Validity

Threats to internal validity

Some change patterns are complex. They span many files and versions. We cannot easily identify them without explanation and documentation from developers. They usually need rigorous analysis based on the relevant code. So we might omit some change patterns.

We collect all the commits from the initialization of projects. The time range of them is very wide. Some changes are not very recent. The development of software is very fast, so some change patterns which are not very recent might not be suitable for the newest software.

Threats to external validity

We analyze 7 Java open-source projects. The number of the projects we select is small compared to the huge amount of open-source projects. These 7 projects may not contains all change patterns in concurrent programming. Concurrent programming is a concept not only in Java but many other programming languages can support concurrent programming. The projects of our study are all written in Java. The change

Feature	Explanation
msgKey	Number of keywords in commit message
file	Number of files in a commit
hunk	Number of hunks in a commit
lineAdd	Number of added lines in a commit
lineRemove	Number of removed lines in a commit
lineSub	lineAdd - lineRemove
lineSum	lineAdd + lineRemove
keyAdd	Number of added keywords in a commit
keyRemove	Number of removed keywords in a commit
keySub	keyAdd - keyRemove
keySum	keyAdd + keyRemove
contextKey	Number of keywords in context code

patterns may be different among different programming languages.

III. RESULTS

A. RQ1. How many types of change patterns in concurrent programming?

Taxonomy There are so many different concurrent related changes in the code history. We can classify them into ? types according to their observed code changes. We first read the commit message to know what this commit does and why. We then examine the added and deleted lines. We catch the concurrent related keywords. We classify changes basically into two big categories. The first is using libraries instead of manual concurrency control. The second is adjustment of concurrency control itself. These types do not contain all the concurrent related code changes. Some changes are infrequent and difficult to classify. It can also be decided by purposes of code changes like eliminating deadlock. But purpose is a subjective concept and it is more difficult to determine the reasons behind the changes.

Table III shows an overview of all types, their explanations and occurrence times. We are going to discuss these change types concretely with examples below.

Changing lock type

Developers switch to different types of locks during the software evolution. We have implicit lock and explicit lock in Java. An implicit lock is denoted by a Java keyword 'synchronized', which can synchronize a block of code as a synchronized block on a monitor object. This keyword can be used to mark both methods and code blocks. It is a special kind of lock for every object has a implicit monitor lock. We do not need to acquire and release locks manually. We do not need to worry about forgetting release locks in a 'finally' block. It is easier for programmers to deal concurrency with 'synchronized' rather than explicit locks. An explicit lock is an implementation of locking in API level. You create a lock object and then you can call the methods of this lock object such as 'lock'. It provides more advanced operations like fairness and condition than implicit lock. Java's standard library provides many locking implementations. Third-party libraries also provide this kind of facilities.

We can also view locks in a different perspective like exclusive and shared locks [33]. This classification is usually used in database system.

The reasons of changes vary in different conditions. When a synchronized block cannot satisfy some advanced requirement like fairness or condition, developers might switch to explicit locks. When they find that they only need a simple exclusive lock, they switch to synchronized block. They also might switch to a reader-writer lock [34] from a normal lock to improve concurrency when there are plenty of concurrent read operations. Here are some examples.

```
1   commit fad9609d13e76e9e3a4e01c96f698bb60b03807e
2   YARN-5825. ProportionalPreemptionalPolicy should use readLock over LeafQueue
        instead of synchronized block. Contributed by Sunil G
3
4   -        synchronized (leafQueue) {
5   +        try {
6   +            leafQueue.getReadLock().lock();
7   // go through all ignore-partition-exclusivity containers first to make
8   // sure such containers will be preemptionCandidates first
9   Map<String, TreeSet<RMContainer>> ignorePartitionExclusivityContainers =
10  @@ -147,6 +148,8 @@
11  preemptAMContainers(clusterResource, selectedCandidates,
        skippedAMContainerlist,
12  resToObtainByPartition, skippedAMSize, maxAMCapacityForThisQueue,
13  totalPreemptionAllowed);
14  +        } finally {
15  +            leafQueue.getReadLock().unlock();
16  }
```

This is a commit of YARN-5825 - `ProportionalPreemptionalPolicy` could use `readLock` over `LeafQueue` instead of a synchronized block. It is a major bug of Hadoop project. They used a synchronized block to synchronize in various places, which can be replaced with a reader lock. There are many different locks which can be used to synchronize. A reader lock is more lightweight than a synchronized block and it allows multiple threads to read simultaneously and hence improves performance under the scenario where most of the operations are reading.

```
1   commit 3e4b1ae6dc786b268505aa2e64067432519c2bcf
2   FRom kkolinko:
3   A ReadWriteLock cannot be used to guard a WeakHashMap.  The
4   WeakHashMap may modify itself on get(), as it processes the reference
5   queue of items removed by GC.
6   Either a plain old lock / synchronization is needed, or some other solution
7   (e.g.  org.apache.tomcat.util.collections.ManagedConcurrentWeakHashMap )
8
9   -        Lock readlock = classLoaderContainerMapLock.readLock();
10  -        try {
11  -            readlock.lock();
12  +        synchronized (classLoaderContainerMapLock) {
13              result = classLoaderContainerMap.get(tcc1);
14  -        } finally {
15  -            readlock.unlock();
16  -        }
```

This is an example in Tomcat. The developer said a `ReadWriteLock` cannot guard a `WeakHashMap`. It is because get() may modify a `WeakHashMap`, in which case a reader lock is not enough. A plain old lock or synchronization should be used. In this example, the synchronization is used finally.

Changing lock instance

You need to create an instance of lock like Java's `ReentrantLock` class first before you can use it. You also need to choose an object to synchronize on when you use synchronized block. Developers may change lock instance during software evolution. It is important to choose the right lock instance to use and decide the order to acquire and release different locks. Sometimes it is hard for developers to do this.

TABLE III
TAXONOMY

Type	Example		Occurrence

Changing lock type	`synchronized (obj) {`
	`    ...`
	`}` ⟹
		`try {`
		`    readLock.lock();`
		`    ...`
		`} finally {`
		`    readLock.unlock();`
		`}`
			4

Changing lock instance	`synchronized (obj1) {`
	`    ...`
	`}` ⟹
		`synchronized (obj2) {`
		`    ...`
		`}`
			6

Changing critical section	`synchronized (obj) {`
	`    code block 1`
	`}` ⟹
		`synchronized (obj) {`
		`    code block 2`
		`}`
			35

Adding or removing `volatile`	`int foo;` ⟹ `volatile int foo;`		47
Thread-safe class replacement	Use thread-safe class instead of handling concurrency control manually		32

No matter explicit locks or synchronized blocks, we always need a lock instance to be acquired and released. There are some best practice like two-phase locking [35], which is a locking method to guarantee serializability. It is a concept originally in database transaction management.

```
1  commit 563e546236217dace58a8031d56d08a27e08160b
2  [FLINK-1419] [runtime] Fix: distributed cache properly synchronized
3
4  public FutureTask<Path> createTmpFile(String name, DistributedCacheEntry entry
       , JobID jobID) {
5  -    synchronized (count) {
6  -       Pair<JobID, String> key = new ImmutablePair<JobID, String>(jobID, name)
            ;
7  -       if (count.containsKey(key)) {
8  -          count.put(key, count.get(key) + 1);
9  +    synchronized (lock) {
10 +       if (!jobCounts.containsKey(jobID)) {
11 +          jobCounts.put(jobID, new HashMap<String, Integer>());
12 +       }
13 +       Map<String, Integer> count = jobCounts.get(jobID);
14 +       if (count.containsKey(name)) {
15 +          count.put(name, count.get(name) + 1);
16        } else {
17 -          count.put(key, 1);
18 +          count.put(name, 1);
19        }
20    }
```

This is a commit of FLINK-1419 - `DistributedCache` doesn't preserver files for subsequent operations, which is a major bug. The description said that it happens that the files are created yet for the operations when subsequent operations are going to access the same file in the `DistributedCache`. They synchronized on `count`, which is a map instance. Instead, they used `lock`, which is an instance of `Object` class. The difference is this instance is only used for synchronization while the former one has its own role not only as a lock. We do not need to blame preference of synchronization usage. This commit also made other changes about synchronization. They modified the critical sections as well.

```
1  commit f0e627bb8c9daedb3b064027cac37ce4849bab64
2  Fix https://bz.apache.org/bugzilla/show_bug.cgi?id=58382
3  Use single object (membersLock) for all locking
4
5  /**
6   * Reset the membership and start over fresh. i.e., delete all the members
7   * and wait for them to ping again and join this membership.
8   */
9  -    public synchronized void reset() {
10 -       map.clear();
11 -       members = EMPTY_MEMBERS ;
12 +    public void reset() {
13 +       synchronized (membersLock) {
14 +          map.clear();
15 +          members = EMPTY_MEMBERS ;
16 +       }
17    }
```

This is another example. The lock instance was originally the instance of the class and now is `membersLock`. They use single object (`membersLock`) for all locking as the commit message said. Using a separated locking instance can allow you to have more precise concurrency control than using synchronized methods.

Changing critical section

Critical section is a code block guarded by synchronization. It is the most common change in terms of concurrency control. We classify these changes into five subtypes: adding synchronization, removing synchronization, adding statements, removing statements, modifing statements. Adding synchronization means to add synchronization to the statements which are not synchronized before. Removing synchronization means to remove synchronization of the statements, which are no more protected now but still exist outside the critical sections. Adding statements means to add new statements to the critical sections. Removing statements means to remove existing statements inside the critical sections. Modifing statements means to modify statements inside the critical sections. The reasons behind the changes depend on specific context.

```
1  commit 17206cc8c21c439d121a66d7c9934cdfa4791a35
2  Fix https://bz.apache.org/bugzilla/show_bug.cgi?id=58386
3  On the basis that access() and finish() are synchronized, extend
       synchronization to other methods that access same fields.
4
5  -    public boolean isAccessed() {
6  +    public synchronized boolean isAccessed() {
7        return this.accessed;
8  }
```

This is an example of adding synchronization. The mothod was not synchronoized before. The basic reason of adding synchronization to existing code is the piece of code might be access concurrently.

```
1  commit c93d9eaf363a535dff25cc4e7db400d879e73bb1
2  Add option to use single actor system for local execution. Use local
       connection manager if a single task manager is used for local execution.
       Remove synchronized blcok in getReceiverList of ChannelManager which
       effectively serialized the connection lookup calls of a single task
       manager.
3
4  -synchronized (this.channelLookup) {
5  -    try{
6  -       lookupResponse = AkkaUtils.<JobManagerMessages.ConnectionInformation>ask(
             channelLookup,
7  -          new JobManagerMessages.LookupConnectionInformation(connectionInfo,
             jobID,
8  -          sourceChannelID), timeout).response();
9  -    }catch(IOException ioe) {
10 -       throw ioe;
11 -    }
12 -}
13 +lookupResponse = AkkaUtils.<JobManagerMessages.ConnectionInformation>ask(
       channelLookup,
14 +    new JobManagerMessages.LookupConnectionInformation(connectionInfo, jobID,
15 +    sourceChannelID), timeout).response();
```

This example is from Flink. Developers remove the synchronization of a code block. If you find the code will not be
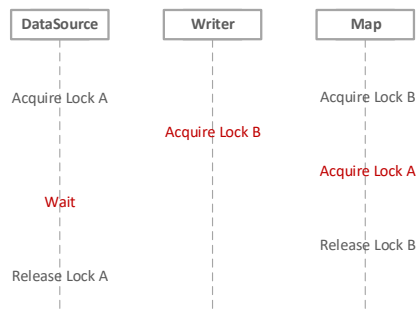
Fig. 1. Deadlock

access concurrently or it can be accessed safely in a concurrent way, you do not need to synchronize the code.

```
1   commit 7e56bfe40589a1aa9b5ef20b342e421823cd0592
2   HDFS-4200. Reduce the size of synchronized sections in PacketResponder.
         Contributed by Suresh Srinivas.
3
4   -    synchronized void enqueue(final long seqno,
5   -        final boolean lastPacketInBlock, final long offsetInBlock) {
6   -      if (running) {
7   -        final Packet p = new Packet(seqno, lastPacketInBlock, offsetInBlock,
8   -          System.nanoTime());
9   -        if(LOG.isDebugEnabled()) {
10  -          LOG.debug(myString + ": enqueue " + p);
11  +    void enqueue(final long seqno, final boolean lastPacketInBlock,
12  +        final long offsetInBlock) {
13  +      final Packet p = new Packet(seqno, lastPacketInBlock, offsetInBlock,
14  +          System.nanoTime());
15  +      if(LOG.isDebugEnabled()) {
16  +        LOG.debug(myString + ": enqueue " + p);
17  +      }
18  +      synchronized(this) {
19  +        if (running) {
20  +          ackQueue.addLast(p);
21  +          notifyAll();
22  +        }
23  -        ackQueue.addLast(p);
24  -        notifyAll();
25         }
26       }
```

This is a commit of HDFS-4200 - Reduce the size of synchronized sections in `PacketResponder`, which is a major improvement. The previous two examples are very easy. This is a complex change compared to previous examples. This example contains multiple types of changes. They change the lock instance, remove synchronization of some code, add some new synchronized code. The developers said the size of synchronized sections can be reduced. It is always meaningful to remove the unnecessary synchronizations. Over-synchronization [18] is a real issue in real-world software.

```
1   commit efca79cfb7b496b4bec70561cc94af069c644ef2
2   [FLINK-2384] [runtime] Move blocking I/O call outside of synchronized block
3   Problem: Waiting on asynchronous write requests with the partition lock can
4   result in a deadlock, because all other operations on the same partition are
5   blocked. It is possible that the I/O writer itself needs to access the
6   partition, in which cases the whole program blocks.
7   Solution: Move the wait outside the synchronized block. This was not necessary
8   before, because no operation assumes the spilling to be finished when the
9   finish call has returned.
10
11    public void finish() throws IOException {
12      synchronized (buffers) {
13        if (add(EventSerializer.toBuffer(EndOfPartitionEvent.INSTANCE))) {
14  -       // If we are spilling/have spilled, wait for the writer to finish.
15  -       if (spillWriter != null) {
16  -         spillWriter.close();
17  -       }
18          isFinished = true;
19        }
20      }
21  +   // If we are spilling/have spilled, wait for the writer to finish.
22  +   if (spillWriter != null) {
23  +     spillWriter.close();
24  +   }
25    }
```

This is an example from Flink. It is a critical bug issue "Deadlock during partition spilling". A user reported the

problem. Here is selected parts of the stack trace.

```
1   "CHAIN DataSource (at createInput(ExecutionEnvironment.java:502) (org.apache.
         flink.api.java.hadoop.mapreduce.HadoopInputFormat)) -> FlatMap (FlatMap
         at readFlinkTuplesFromThriftParquet(ParquetThriftEntitons.java:96))
         (7/8)" daemon prio=10 tid=0x00007f934005b000 nid=0x73c4 in Object.wait()
         [0x00007f93c16ac000]
2   java.lang.Thread.State: TIMED_WAITING (on object monitor)
3
4   "IOManager writer thread #1" daemon prio=10 tid=0x00007f93d8b7b000 nid=0x73a8
         waiting for monitor entry [0x00007f93c2fc5000]
5   java.lang.Thread.State: BLOCKED (on object monitor)
6
7   "Map (Projection [0, 1, 2, 3, 4]) (7/8)" daemon prio=10 tid=0x00007f92b0434800
         nid=0x74a3 waiting for monitor entry [0x00007f93a32f1000]
8   java.lang.Thread.State: BLOCKED (on object monitor)
```

The `DataSource` is waiting to be notified by the `Writer` when holding lock A. The `Writer` is trying to acquire lock B. But lock B is held by the `Map`. And the `Map` is trying to acquire lock A, which is held by the `DataSource`. Here is a waiting chain. This is a kind of bug known as deadlock. The problem is that the `DataSource` does not need to hold lock A while waiting to be notified. So the developer removed the unnecessary synchronization.

Adding or removing volatile

`volatile` is a Java keyword, which is used to mark a variable which should be saved in main memory so that every thread which reads the variable can read the latest value from the main memory not the outdated value in the CPU cache. This provides visibility of variables across multiple threads. It also guarantees the happen-before relation. Using `volatile` correctly instead of locking can improve the performance in terms of concurrency. But this keyword is subtle. Developers need careful reasoning when dealing with `volatile`. Sometimes `volatile` is not enough. There may be race conditions when multiple thread read and write `volatile` fields simultaneously.

```
1   commit 8313fa0f1ca277e9633a78f461804abc3c5515b8
2   Fix https://bz.apache.org/bugzilla/show_bug.cgi?id=58392
3   Double-checked locking needs to use volatile to be thread-safe
4
5   -    protected Membership membership = null;
6   +    protected volatile Membership membership = null;
```

It is a commit for bug 58392 in Bugzilla. It is reported by a race detector that there is data race on field. Double-checked locking is a synchronization pattern in software engineering. It first check the condition without lock to reduce the time overhead when the condition is not satisfied. A typical usage of it is singleton pattern which uses lazy initialization to provide an unique instance during the process execution time in multi-threaded scenario. But sometimes programmers make some mistakes using this pattern like omitting `volatile` modifier of a class member. If `volatile` is not used, another thread which gets into the method may observe that `membership` has been assigned but the real initialization of the object has not been done yet due to the JVM's instructions reordering. Some work has been done in double-checked locking pattern [36], which can use this pattern automatically.

```
1   commit 560cd00890b3f6af2aca0c3a9d51a45f880692dd
2   Fix a FindBugs warning (increment of volatile not atomic)
3
4   -    private volatile int requestCount;
5   -    private volatile int errorCount;
6   +    private final AtomicInteger requestCount = new AtomicInteger(0);
7   +    private final AtomicInteger errorCount = new AtomicInteger(0);
8   ...
9   -        requestCount++;
10  +        requestCount.incrementAndGet();
```

This is an example in Tomcat. The developer used FindBugs to check the code and it said increment of a `volatile` field is not atomic. This is a wrong demonstration of how to use `volatile`. The original code is not safe because increment is not an atomic operation. It includes read, calculate and write operations to complete the increment. So the developer used a thread-safe class instead.

Thread-safe class replacement

Thread-safe class replacement is an adoption of thread-safe class instead of handling the concurrency control by yourself. This type of changes is a very common in practice. There are two major advantages to apply thread-safe classes. Firstly, it simplifies the code. Developers usually need to write less code by simply calling APIs than implementing on their own. Secondly, these classes are mostly carefully designed by experienced class authors who are good at concurrent programming. So these classes can offer not only better correctness and robustness, but also possible performance improvement. Developers can spend less time in writing and debugging their code when employing thread-safe classes. This type of changes can improve both the efficiency and quality of software development. For example, developers can employ thread-safe containers instead of adding synchronizations with use of non-thread-safe containers.

```
1   commit a258263ecfa1d9efe03761f5e3b73e8e6ddb4a43
2   HDFS-4029. GenerationStamp should use an AtomicLong. Contributed by Eli
        Collins
3
4   -   private volatile long genstamp;
5   +   private AtomicLong genstamp = new AtomicLong();
6   ...
7   -   public synchronized long nextStamp() {
8   -       this.genstamp++;
9   -       return this.genstamp;
10  +   public long nextStamp() {
11  +       return genstamp.incrementAndGet();
12      }
```

This commit is from hadoop. It is a fix of issue HDFS-4029 "GenerationStamp should use an AtomicLong" whose priority is major. The code synchronize the method nextStamp for it might be invoked concurrently. Method nextStamp increases genstamp by one and then return it. The developers found that it would be cleaner to use an AtomicLong so that genstamp itself is atomic and they do not have to synchronize the various accesses to it. AtomicLong is a thread-safe version of type long. It allows users to update it atomically without any synchronization. Its internal implementation is not using synchronized method or block. It uses sun.misc.Unsafe which provides many unsafe but fast operations.

```
1   commit 7f443f67eaa588323f912f3922cff9b699b38fbd
2   LUCENE-2779: Use ConcurrentHashMap in RAMDirectory
3
4   -   protected HashMap<String,RAMFile> fileMap = new HashMap<String,RAMFile>();
5   +   protected Map<String,RAMFile> fileMap = new ConcurrentHashMap<String,
        RAMFile>();
6   ...
7   @Override
8   public final boolean fileExists(String name) {
9       ensureOpen();
10  -   RAMFile file;
11  -   synchronized (this) {
12  -       file = fileMap.get(name);
13  -   }
14  -   return file != null;
15  +   return fileMap.containsKey(name);
16  }
```

This commit is from lucene-solr. It is a commit for LUCENE-2779 which is a minor-priority improvement. It is better to use a thread-safe version collection Concurren-

TABLE IV
TOP CLASSES

Class	#Add	Class	#Del
AtomicInteger	2780	AtomicInteger	4111
AtomicLong	1701	AtomicBoolean	2504
CountDownLatch	1698	ConcurrentHashMap	2415
AtomicBoolean	1676	AtomicLong	2225
ConcurrentHashMap	1561	CountDownLatch	1513
AtomicReference	1030	AtomicReference	1224
Executors	921	Executors	1105
LinkedBlockingQueue	689	ThreadPoolExecutor	1034
ConcurrentLinkedQueue	638	LinkedBlockingQueue	864
ThreadPoolExecutor	583	ConcurrentLinkedQueue	797

tHashMap instead of using HashMap and synchronizing the access code. This thread-safe class not only simplify the way of using a hash map, but also improve the performance compared to manual synchronization like the example because the implementation of ConcurrentHashMap is carefully designed. Retrieval operations do not acquire any locks at all and update operations do not lock the entire map. It supports full concurrency of retrievals and high expected concurrency for updates.

B. RQ2. How frequent do concurrent related code modification appear in different kinds of Java open-source projects?

Figure 1 shows the numbers of concurrent related commits and all commits of each month in all projects of our study. It also shows the percentage of concurrent related commits. Each subfigure has two subfigures inside. The x axis represents the time in month. The upper subfigure has two lines. The higher line shows the number of all commits while the lower line shows the number of concurrent related commits. The number of concurrent related commits is relatively small compared to the number of all commits. The two indexes have a positive correlation generally. The bottom subfigure shows the percentage of concurrent related commits. The percentages differ in project and time. For example, the percentage of concurrent related commits in mahout is relatively lower than other projects. The percentage in Hadoop is stable and high compared to other projects.

C. RQ3. What is the trend of concurrent programming classes usage statistically?

We write a program to count and analyze concurrent programming classes usage. Table IV shows top 10 classes added and deleted in the history. Some classes are both active in the added and the deleted column like AtomicInteger and CountDownLatch. This is not surprising because a deletion of class does not mean this class is abandoned. This also indicates this class is active. An interesting observation is that deletions appear more than addition.

We draw line for addition and removal of each class. Figure 3 and figure 4 show examples. We can see that both numbers of addition and removal increase as a whole as time passing by. This indicates that this class is attracting developers' attention.
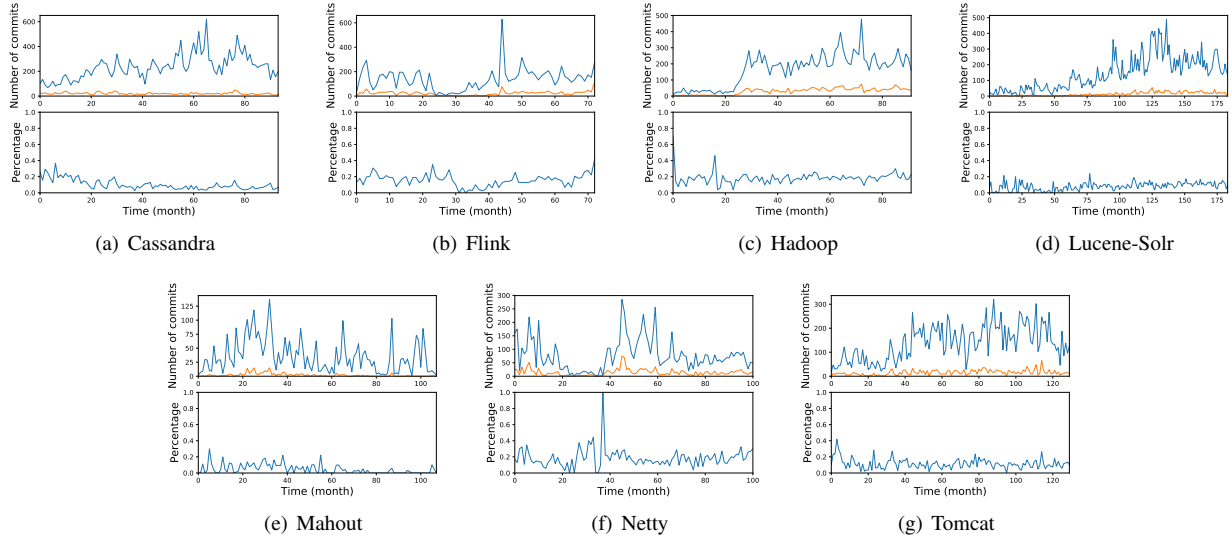
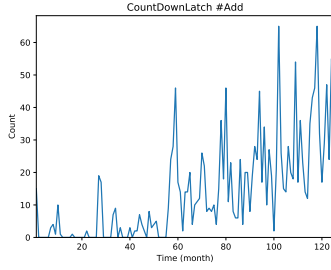Fig. 2. Number of concurrent related commits compared to all commits
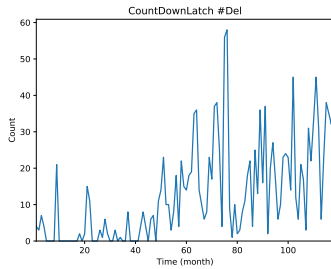

Fig. 3. Addition of CountDownLatch

Fig. 4. Removal of CountDownLatch

D. RQ4. Can these change patterns be applied to real world projects?

The answer is yes. We have found some contexts which are suitable for the change patterns in different projects.

```
1    +import java.util.concurrent.ThreadLocalRandom;
2     public class DRandom {
3    -     private static ThreadLocal<Random> random = new ThreadLocal<Random>() {
4    -         protected Random initialValue() {
5    -             return new Random();
6    -         }
7    -     };
8         public static Random get() {
9    -         return random.get();
10   +         ThreadLocalRandom.current();
11        }
12    }
```

This example shows a change from ThreadLocal<Random> to ThreadLocalRandom from JDK7. It is from a Schmince-2, a game on Google Play. ThreadLocalRandom should be used when available in place of ThreadLocal<Random>. For JDK7 the difference is minimal, but JDK8 starts including optimizations for ThreadLocalRandom. The ThreadLocal class provides thread-local variables. The ThreadLocalRandom class is a random number generator isolated to the current thread.

IV. DISCUSSION

We also have some interesting findings in collecting and analyzing the concurrent related code changes.

(1) Some changes are contrary. Different developers may modify their code in an opposite direction. Here is an example.

```
1    commit f5fab1f64ba1le04e52bd6251ca62fc854e9578c
2    Whoops. Fix regression in r1724015.
3    Code was used although I can't see why a simple AtomicInteger wasn't
         sufficient.
4
5    +    private final AtomicInteger aprPoolDestroyed = new AtomicInteger(0);
6    -    private static final AtomicIntegerFieldUpdater<OpenSSLContext>
         DESTROY_UPDATER = AtomicIntegerFieldUpdater.newUpdater(OpenSSLContext.
         class, "aprPoolDestroyed");
```

A previous commit switched to `AtomicInteger` from `AtomicIntegerFieldUpdater`. But now this developer reverse the change. In this example, `AtomicIntegerFieldUpdater` is a class which enables atomic updates to `volatile` field of classes. We can see that developers in one project may have divergence in a problem.

(2) Developers are using some code-checking tools like FindBugs to help them inspect their code, but sometimes these tools are not enough.

The examples above show that some developers are using code checking tools like FindBugs. Some tools are useful in development environments [37]. But these kind of tools don't help developers correct and eliminate the warnings automatically. One developer in Tomcat said "make the volatile

anyway so FindBugs doesn't complain" in the commit log. This indicates code checking tools still have room to improve.

This research provides some implications from different kinds of perspectives.

(1) Developers are facing more and more concurrent programming requirements now. But concurrent programming is notoriously error-prone because of the complexity of data synchronization and thread interleaving. Our study gives developers some guidelines of writing concurrent programs. First, use handy concurrent libraries to finish the job instead of rewrite them by yourself unless all the available concurrent libraries cannot satisfy your requirement and you are absolutely confident of your comcurrent programming skills. Using existing libraries allows you to write less code to finish the same work and enjoy the high quality of implementation which is always reliable, strong and fast. Second, always switch to new-version libraries because they usually provide higher performance and robustness.

(2) Automatic tools are needed to help developers inspect and revise concurrent programs with the help of history information. There has already been some tools, but they usually look for concurrent bugs such as race detection, deadlock detection and atomicity violation without considering software evolution history. Both project specific and project independent transformation patterns exist in real-world software projects. So we need some concurrent code refactoring tools to give advice of what code need change and perform the transformations automatically. It is a chance for IDE manufacturer to make the IDE more intelligent in inspecting and modifying the code. Developers will benefit a lot if such kind of automatic tools can actually help them automate their development and maintaining activities.

V. RELATED WORK

Studies on concurrent programming Concurrent programming attract many researchers' attention. Pinto *et al*. [1] studied the usage of Java's concurrency constructs from 2227 real world, Java projects. They found that more than 75% of projects employ concurrency control or create threads. Similarly, Wu *et al*. [27] conducted a study on C++ concurrency constructs. Okur and Dig [28] studied how developers use parallel libraries. They analyzed 655 open-source applications developed by 1609 programmers. The applications adopted Task Parallel Library and Parallel Language Integrated Query, which are Microsoft's parallel libraries in C#. They reveal some interesting facts such as at least 10% of programmers misuse the two libraries so that the programs run sequentially rather than concurrently. David *et al*. [10] investigated questions people wanted to know about synchronization from hardware to high-level software. They found that scalability of synchronization is a main property of hardware. Pinto *et al*. [11] conducted a study on the most popular questions of concurrent programming from StackOverflow. They found that most of questions asked about concurrent programming are basic concepts such as "what is a mutex?". Sadowski *et al*. [38] studied data races evolution from history of two open-source

projects. They found that many data races exist in most time of the projects' history. Xin *et al*. [39] conducted an empirical study on lock usage including lock manifestation and lock usage. They found that most functions which use locks only acquire one lock. Lu *et al*. [4] studied characteristics of real world concurrency bugs. They had some findings such as one third of the non-deadlock concurrency bugs involve more than one variable and current detection tools cannot handle them well. Zhang *et al*. [40] presented a lightweight system to detect and tolerate concurrency bugs. It can detect a large range of concurrency bugs and the overhead is negligible.

Program transformation Many studies have been conducted to understand program transformation and to transform programs automatically. Okur *et al*. [41] studied 880 open-source C# projects and found that converting code from low-level to high-leveler parallel abstractions is tedious and error-prone. They presented two refactoring tools to help developers with the migration. Lin *et al*. [42] studied 104 open-source applications to understand how AsyncTask is used, underused and misused in practice. They found that hundreds of places have the chances to apply AsyncTask and nearly half of the refactoring is done manually. They presented Asynchronizer, a refactoring tool which can extract code into AsyncTask automatically. Meng *et al*. [20] presented Sydit, a tranformation tool which can tranform programs from examples. Sydit can make the similar edits to what human make on 70% (39/56) cases.

Identification of commits

VI. CONCLUSION

We conduct a study on change patterns in concurrent programming. We select concurrent related commits by textual analysis and machine learning. We examine the code changes to understand what the changes are and why they change. We identify some change patterns and establish a taxonomy of these change patterns. The most common types of change patterns are changing critical sections, adding or removing `volatile` and thread-safe class replacement. We also find some interesting findings.

REFERENCES

[1] G. Pinto, W. Torres, B. Fernandes, F. C. Filho, and R. S. M. de Barros, "A large-scale study on the usage of java's concurrent programming constructs," *Journal of Systems and Software*, vol. 106, pp. 59–81, 2015.

[2] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a)," *CoRR*, vol. abs/1701.00854, 2017.

[3] H. Sutter and J. R. Larus, "Software and the concurrency revolution," *ACM Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[4] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.

[5] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," in *PLDI*, 2009.

[6] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter, "Sound static deadlock analysis for c/pthreads," in *ASE*, 2016.

[7] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," in *PLDI*, 2008.

[8] D. Lea, "A java fork/join framework," in *Java Grande*, 2000.

[9] M. Bagherzadeh, "Panini: a concurrent programming model with modular reasoning," in *SPLASH*, 2015.

[10] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *SOSP*, 2013.

[11] G. Pinto, W. Torres, and F. Castor, "A study on the most popular questions about concurrent programming," in *PLATEAU@SPLASH*, 2015.

[12] H. Borges, "On the popularity of github software," in *ICSME*.

[13] H. Borges, A. C. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *ICSME*.

[14] J. Kim, D. S. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *ICSE*, 2016.

[15] M. Wahler, U. Drofenik, and W. Snipes, "Improving code maintainability: A case study on the impact of refactoring," in *ICSME*, 2016.

[16] Q. Jiang, X. Peng, H. Wang, Z. Xing, and W. Zhao, "Summarizing evolutionary trajectory by grouping and aggregating relevant code changes," in *SANER*, 2015.

[17] N. Meng, M. Kim, and K. S. McKinley, "LASE: locating and applying systematic edits by learning from examples," in *ICSE*.

[18] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu, "What change history tells us about thread synchronization," in *ESEC/FSE*, 2015.

[19] G. Santos, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "System specific, source code transformations," in *ICSME*, 2015.

[20] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," in *PLDI*.

[21] M. Samak and M. K. Ramanathan, "Trace driven dynamic deadlock detection and reproduction," in *PPoPP*, 2014.

[22] M. Eslamimehr and J. Palsberg, "Sherlock: scalable deadlock detection for concurrent programs," in *FSE*, 2014.

[23] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond, "Doublechecker: efficient sound and precise atomicity checking," in *PLDI*, 2014.

[24] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[25] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE*, 2009.

[26] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with AST analysis," in *ICSME*, 2013.

[27] D. Wu, L. Chen, Y. Zhou, and B. Xu, "An extensive empirical study on C++ concurrency constructs," *Information & Software Technology*, vol. 76, pp. 1–18, 2016.

[28] S. Okur and D. Dig, "How do developers use parallel libraries?" in *FSE*.

[29] J. Loeliger, *Version Control with Git - Powerful techniques for centralized and distributed project management*. O'Reilly, 2009.

[30] Y. Tian, J. L. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *ICSE*, 2012.

[31] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[32] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[33] Z. M. Kedem and A. Silberschatz, "Locking protocols: From exclusive to shared locks," *J. ACM*, vol. 30, no. 4, pp. 787–804, 1983.

[34] P. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"," *Commun. ACM*, vol. 14, no. 10, pp. 667–668, 1971.

[35] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[36] K. Ishizaki, S. Daijavad, and T. Nakatani, "Transforming java programs for concurrency using double-checked locking pattern," in *ISPASS*, 2014.

[37] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using findbugs on production software," in *OOPSLA Companion*, 2007.

[38] C. Sadowski, J. Yi, and S. Kim, "The evolution of data races," in *MSR*, 2012.

[39] R. Xin, Z. Qi, S. Huang, C. Xiang, Y. Zheng, Y. Wang, and H. Guan, "An automation-assisted empirical study on lock usage for concurrent programs," in *ICSM*, 2013.

[40] M. Zhang, Y. Wu, S. Lu, S. Qi, J. Ren, and W. Zheng, "A lightweight system for detecting and tolerating concurrency bugs," *IEEE Trans. Software Eng.*

[41] S. Okur, C. Erdogan, and D. Dig, "Converting parallel code from low-level abstractions to higher-level abstractions," in *ECOOP*, 2014.

[42] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for android applications through refactoring," in *FSE*.