Bare Demo of IEEEtran.cls for IEEE Conferences

Michael Shell
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: http://www.michaelshell.org/contact.html

Homer Simpson
Twentieth Century Fox
Springfield, USA
Email: homer@thesimpsons.com

James Kirk
and Montgomery Scott
Starfleet Academy
San Francisco, California 96678–2391
Telephone: (800) 555–1212
Fax: (888) 555–1212

***Abstract*—The abstract goes here.**

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].

Rui 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 libirary usage, thread resource management. Study on critical sections are not enough to understand real-world thread synchronization.

Gustavo 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. Numerous commits are submited every day. We want to know not only the usage of concurrent programming, but also the trend of it.

Gustavo 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.

We studied concurrent programs from a perspective of software evolution history and found 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 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. They can learn from existing change patterns. 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.

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.

Our main contributions are:

- We identify and classify change patterns in concurrent programs from 102028 commits of 7 open-source projects. The most common types of change patterns are thread-safe class replacement, synchronization addition, synchronization removal.
- We observe some interesting findings: .
- We find some contexts to which we can apply the patterns in real-world projects. We make some pull requests and some of them are accepted.
- We give some inspirations to concurrent program or library developers and analysis tool developers.

The rest of paper is organized as follows: Section 2 presents the methodology of our study. Section 3 presents our result and 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, Tomcat, Cassandra, Lucene-solr, Netty, Flink and Mahout as shown in Table 1. 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 [20] 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.

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

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?

Researchers found that many code changes are similar [21]. Similar changes of code can be extracted into change patterns [22]. 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],[23],[24] 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, language level constructs like synchronized and volatile 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. Is there any context in real-world projects to which we can apply these change patterns?

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 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. Yuan Tian et al. gave a successful example of identifing bug fixing patches using machine learning[25]. 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.

TABLE II
FEATURES OF DATA

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

We use the SVM[26] 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[27]. 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.

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. 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

TABLE III
TAXONOMY

Name	Explanation	Occurrence
Changing lock type	Switch to another type of lock. We have implicit monitor lock, ReentrantLock and so on	
Changing lock instance		
Addition or removing synchronization		
Addition or removing volatile modifier	Add volatile modifier to a field of class	
Thread-safe class replacement	Use thread-safe class instead of handling concurrency control manually	
Other class replacement	Some other concurrent related class replacement	

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 [28]. 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 [29] from a normal lock to improve concurrency when there are plenty of concurrent read operations. Here are some examples.

```
commit fad9609d13e76e9e3a4e01c96f698bb60b03807e
YARN-5825. ProportionalPreemptionalPolicy should use readLock over LeafQueue
       instead of synchronized block. Contributed by Sunil G

-      synchronized (leafQueue) {
+      try {
+        leafQueue.getReadLock().lock();
// go through all ignore-partition-exclusivity containers first to make
// sure such containers will be preemptionCandidates first
Map<String, TreeSet<RMContainer>> ignorePartitionExclusivityContainers =
@@ -147,6 +148,8 @@
preemptAMContainers(clusterResource, selectedCandidates,
       skippedAMContainerlist,
resToObtainByPartition, skippedAMSize, maxAMCapacityForThisQueue,
totalPreemptionAllowed);
+      } finally {
+        leafQueue.getReadLock().unlock();
}
```

This is a commit of YARN-5825 - ProportionalPreemptionalPolicy could use readLock over LeafQueue instead of 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.

```
commit 3e4b1ae6dc786b268505aa2e64067432519c2bcf
FRom kkolinko:
A ReadWriteLock cannot be used to guard a WeakHashMap.  The
WeakHashMap may modify itself on get(), as it processes the reference
queue of items removed by GC.
Either a plain old lock / synchronization is needed, or some other solution
(e.g.  org.apache.tomcat.util.collections.ManagedConcurrentWeakHashMap )
```

This is an example in Tomcat. The developer said a ReadWriteLock cannot guard a WeakHashMap. A plain old lock or synchronization should be used.

Changing lock instance

You need to create an instance of lock like Java's Reentrant 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.

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 [30], which is a locking method to guarantee serializability. It is a concept originally in database transaction management.

```
commit 563e546236217dace58a8031d56d08a27e08160b
[FLINK-1419] [runtime] Fix: distributed cache properly synchronized

public FutureTask<Path> createTmpFile(String name, DistributedCacheEntry entry
       , JobID jobID) {
-    synchronized (count) {
-      Pair<JobID, String> key = new ImmutablePair<JobID, String>(jobID, name)
       ;
-      if (count.containsKey(key)) {
-        count.put(key, count.get(key) + 1);
+    synchronized (lock) {
+      if (!jobCounts.containsKey(jobID)) {
+        jobCounts.put(jobID, new HashMap<String, Integer>());
+      }
+      Map<String, Integer> count = jobCounts.get(jobID);
+      if (count.containsKey(name)) {
+        count.put(name, count.get(name) + 1);
      } else {
-        count.put(key, 1);
+        count.put(name, 1);
      }
}
```

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.

```
commit f0e627bb8c9daedb3b064027cac37ce4849bab64
Fix https://bz.apache.org/bugzilla/show_bug.cgi?id=58382
Use single object (membersLock) for all locking

/**
* Reset the membership and start over fresh. i.e., delete all the members
* and wait for them to ping again and join this membership.
*/
-    public synchronized void reset() {
-        map.clear();
-        members = EMPTY_MEMBERS ;
+    public void reset() {
+        synchronized (membersLock) {
+            map.clear();
+            members = EMPTY_MEMBERS ;
+        }
}
```

This is another example. The lock variable is originally the monitor of the object and now is membersLock. They turned to use single object (membersLock) for all locking as the commit message said.

Changing critical sections

Adding synchronization means to add synchronization to the code block which are not synchronized before.

```
commit 17206cc8c21c439d121a66d7c9934cdfa4791a35
Author: Mark Thomas <markt@apache.org>
Date:   Wed Sep 16 13:37:35 2015 +0000

Fix https://bz.apache.org/bugzilla/show_bug.cgi?id=58386
On the basis that access() and finish() are synchronized, extend
       synchronization to other methods that access same fields.

-    public boolean isAccessed() {
+    public synchronized boolean isAccessed() {
return this.accessed;
}
```

Synchronization removal is a removal of critical section, which is usually not allowed to be executed by multiple threads.

```
commit 7e56bfe40589a1aa9b5ef20b342e421823cd0592
Author: Suresh Srinivas <suresh@apache.org>
Date:   Mon Nov 26 20:47:58 2012 +0000

HDFS-4200. Reduce the size of synchronized sections in PacketResponder.
       Contributed by Suresh Srinivas.

-    synchronized void enqueue(final long seqno,
-        final boolean lastPacketInBlock, final long offsetInBlock) {
-      if (running) {
-        final Packet p = new Packet(seqno, lastPacketInBlock, offsetInBlock,
-            System.nanoTime());
-        if(LOG.isDebugEnabled()) {
-          LOG.debug(myString + ": enqueue " + p);
+    void enqueue(final long seqno, final boolean lastPacketInBlock,
+        final long offsetInBlock) {
+      final Packet p = new Packet(seqno, lastPacketInBlock, offsetInBlock,
+          System.nanoTime());
+      if(LOG.isDebugEnabled()) {
+        LOG.debug(myString + ": enqueue " + p);
+      }
+      synchronized(this) {
+        if (running) {
+          ackQueue.addLast(p);
+          notifyAll();
}
-        ackQueue.addLast(p);
-        notifyAll();
      }
    }
```

This is a commit of HDFS-4200 - Reduce the size of synchronized sections in PacketResponder. It is a major improvement. 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.

```
commit efca79cfb7b496b4bec70561cc94af069c644ef2
Author: Ufuk Celebi <uce@apache.org>
Date:   Thu Jul 23 15:19:57 2015 +0200

[FLINK-2384] [runtime] Move blocking I/O call outside of synchronized block
Problem: Waiting on asynchronous write requests with the partition lock can
result in a deadlock, because all other operations on the same partition are
blocked. It is possible that the I/O writer itself needs to access the
partition, in which cases the whole program blocks.
Solution: Move the wait outside the synchronized block. This was not necessary
before, because no operation assumes the spilling to be finished when the
finish call has returned.

public void finish() throws IOException {
synchronized (buffers) {
if (add(EventSerializer.toBuffer(EndOfPartitionEvent.INSTANCE))) {
-     // If we are spilling/have spilled, wait for the writer to finish.
-     if (spillWriter != null) {
-       spillWriter.close();
-     }
isFinished = true;
}
}
+
+ // If we are spilling/have spilled, wait for the writer to finish.
+ if (spillWriter != null) {
+   spillWriter.close();
+ }
}
```

This is an example from flink. It is a critical bug issue "Deadlock during partition spilling". A user reported the problem. The developer wrote a detailed message, which describes the problem, reason and solution.

Adding or removing volatile modifier

Volatile addition is a addition of the volatile keyword. Volatile is a keyword of Java. It is used to mark a variable which should be saved in main memory. Any read or write operation of the variable should visit the main memory instead of only using cache. This provides visibility of the latest value of variable across multiple threads.

```
commit 8313fa0f1ca277e9633a78f461804abc3c5515b8
Author: Mark Thomas <markt@apache.org>
Date:   Thu Sep 17 09:26:08 2015 +0000

Fix https://bz.apache.org/bugzilla/show_bug.cgi?id=58392
Double-checked locking needs to use volatile to be thread-safe
...
-    protected Membership membership = null;
+    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 forgetting 'volatile' modifier of the member of object in Java.

Volatile addition is a removal of the volatile keyword.

Thread-safe class replacement

Thread-safe class replacement is an adoption of thread-safe class instead of handling the concurrency control by yourself. It is a very common category.

```
commit a258263ecfa1d9efe03761f5e3b73e8e6ddb4a43
Author: Eli Collins <eli@apache.org>
Date:   Wed Oct 17 04:58:24 2012 +0000

HDFS-4029. GenerationStamp should use an AtomicLong. Contributed by Eli
       Collins

...
-  private volatile long genstamp;
+  private AtomicLong genstamp = new AtomicLong();
...
-  public synchronized long nextStamp() {
-    this.genstamp++;
-    return this.genstamp;
+  public long nextStamp() {
+    return genstamp.incrementAndGet();
}
...
```

This commit is from hadoop. "..." represents that some code are omitted. 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.

```
commit 7f443f67eaa588323f912f3922cff9b699b38fbd
Author: Shai Erera <shaie@apache.org>
Date:   Mon Nov 29 15:07:41 2010 +0000

LUCENE-2779: Use ConcurrentHashMap in RAMDirectory
...
-  protected HashMap<String,RAMFile> fileMap = new HashMap<String,RAMFile>();
+  protected Map<String,RAMFile> fileMap = new ConcurrentHashMap<String,
       RAMFile>();
...
@Override
public final boolean fileExists(String name) {
```

```
       ensureOpen();
-      RAMFile file;
-      synchronized (this) {
-        file = fileMap.get(name);
-      }
-      return file != null;
+      return fileMap.containsKey(name);
   }
...
```

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 ConcurrentHashMap 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. It supports full concurrency of retrievals and high expected concurrency for updates.

Other class replace

Thread resource management

When we do concurrent programming, we need to pay attention to resource management such as threads, locks.

Thread management is to deal with the management of thread-related resources.

Thread sleep wait notify

It is a another way of synchronization which is less common than locking.

Final in multiple threads

```
commit 470c87dbc6c24dd3b370f1ad9e7ab1f6dabd2080
Author: Colin Patrick Mccabe <cmccabe@cloudera.com>
Date:   Tue May 19 10:49:17 2015 -0700

HADOOP-11970. Replace uses of ThreadLocal<Random> with JDK7 ThreadLocalRandom
       (Sean Busbey via Colin P. McCabe)
...
-import java.util.Random;
+import java.util.concurrent.ThreadLocalRandom;
...
-  private static ThreadLocal<Random> RANDOM = new ThreadLocal<Random>() {
-    @Override
-    protected Random initialValue() {
-      return new Random();
-    }
-  };
...
-       final double ratio = RANDOM.get().nextDouble() + 0.5;//0.5 <= ratio
        <=1.5
+       // ensure 0.5 <= ratio <=1.5
+       final double ratio = ThreadLocalRandom.current().nextDouble() + 0.5;
```

This example shows a switch from ThreadLocal<Random> to ThreadLocalRandom from JDK7. It is of the issue HADOOP-11970 which is a major improvement. This issue says that 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 since 1.7.

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 in mahout is relatively lower than other projects.

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

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.

D. RQ4. Is there any context in real-world projects to which we can apply these change patterns?

E. Threats to Validity

All the change patterns are summarized from real commits of projects. Different developers may have different taste and preference. Their behavior on similar conditions may be different sometimes contradictory. We indeed find that some changes are contradictory.

Some change patterns are not easy to determine the right occasion to apply especially those concurrency control problems like dead lock, race condition. They usually need rigorous analysis based on the dependent code.

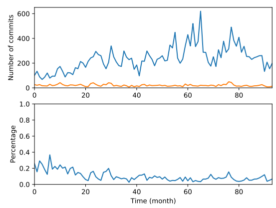
We collect all the commits from the initialization of projects. The time range of them is very wide. Some changes are not very new. The development of software is very fast, so some change patterns which are not very new might not be suitable for the newest software.

IV. DISCUSSION

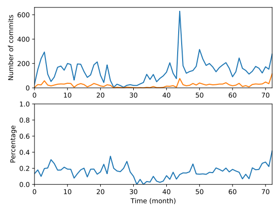
We also have some interesting findings in collecting and analyzing the concurrent related code changes.

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

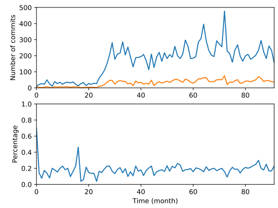
```
commit f5fab1f64ba11e04e52bd6251ca62fc854e9578c
Whoops. Fix regression in r1724015.
Code was used although I can't see why a simple AtomicInteger wasn't
       sufficient.

+     private final AtomicInteger aprPoolDestroyed = new AtomicInteger(0);
```



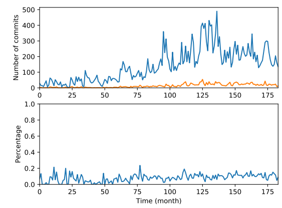
(a) Cassandra



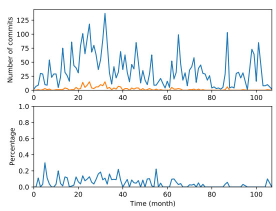
(b) Flink



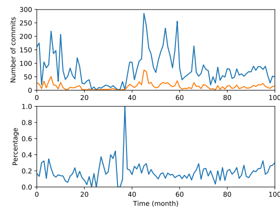
(c) Hadoop



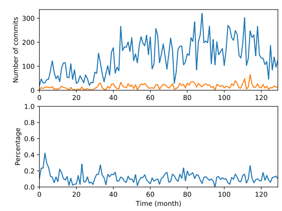
(d) Lucene-Solr



(e) Mahout



(f) Netty



(g) Tomcat

Fig. 1. Number of concurrent related commits compared to all commits

```
-    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. AtomicIntegerFieldUpdater is a class which enables atomic updates to volatile field of classes.

(2) Developers are using some code-checking tools like findbugs to help them inspect their code.

The examples above show that some developers are using code checking tools like findbugs. Some tools are useful in development environments [31]. 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

Empirical studies on concurrent programming Concurrent programming attract many researchers' attention. Many empirical studies have been done. Tudor David et al [10] investigated everything about synchronization people wanted to know but not dared to ask. Semih Okur and Danny Dig [24] 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, and developers are making their code too complex but it is unnecessary.

Program transformation

VI. Conclusion

We conduct a study on change patterns in concurrent programming. We find many change patterns and establish a taxonomy of these change patterns.

Acknowledgment

The authors would like to thank anyone who might have helped us with this study.

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

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

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

[23] 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.

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

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

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

[27] 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.

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

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

[30] 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.

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