How do Programmers Maintain Concurrent Code

Feiyue Yu, Hao Zhong, Beijun Shen
Department of Computer Science, Shanghai Jiao Tong University, China
{yufeiyue, zhonghao, bjshen}@sjtu.edu.cn

***Abstract*—Concurrent programming is pervasive in nowadays software development. Many programmers believe that concurrent programming is difficult, and maintaining concurrency code is error-prone. Although researchers have conducted empirical studies to understand concurrent programming, they still rarely study how programmers maintain concurrent code. To the best of our knowledge, only a recent study explored the modifications on critical sections, and many related questions are still open. In this paper, we conduct an empirical study to explore how programmers maintain concurrent code. We analyze more concurrency-related commits and explore more issues such as the change patterns of maintaining concurrent code than the previous study. We summarize five change patterns according to our analysis on 696 concurrency-related commits. We apply our change patterns to three open source projects, and synthesize three pull requests. Until now, two of them have been accepted. Our results can be useful for programmers to maintain concurrent code and for researchers to implement treating techniques.**

I. Introduction

Many practitioners and researchers believe that the software maintenance phase is one of the most expensive phases, in the life cycle of a software system. Some reports (*e.g.*, [1]) claim that the software maintenance phase accounts for almost 80% of the whole budget. With the maintenance of software, many revision histories are accumulated [3]. Based on such revision histories, researchers have conducted various empirical studies to understand how programmers maintain code (*e.g.*, evolution of design patterns [2], fine-grained modifications [9], and the evolution of APIs [14]). These empirical studies deepen our understanding on software maintenance, and provide valuable insights on how to maintain code in future development.

In recent years, to fully leverage the potential of multi-core CPUs, concurrent programming has become increasingly popular [18]. For example, Pinto *et al.* [18] investigated 2,227 projects, and their results show that more than 75% of these projects employ some concurrency control mechanism. Despite of its popularity, many programmers find that concurrent programming is difficult [15], and often introduce relevant bugs in their code [13]. Thorough empirical studies on how programmers maintain such code are needed. However, this topic is still rarely explored. To the best of our knowledge, only a recent study [10] was conducted to understand how programmers maintain concurrent code. Although the study is insightful and explores many aspects of concurrent programming, it is still incomplete. Their study sampled only 25 concurrency-related commits, and focuses on limited topics such as over synchronization and how concurrency bugs originate. As a result, many relevant questions are still open. For example, are there any patterns, when programmers maintain concurrent code? Indeed, such patterns are useful for programmers when they maintain code. For example, Santos *et al.* [20] have explored the change patterns during software maintenance, and their results show that extracted change patterns can be applied to new code locations. However, their study does not touch the change patterns of concurrent code. A more detailed analysis can have the following benefits:

Benefit 1. The results can deepen the knowledge on how to maintain concurrent code. Due to the complexity of concurrent programming, we find that even experienced developers can be confused when they maintain relevant code. For example, Mark Thomas is a member of the Apache Tomcat Project Management Committee[1], and senior software engineer at the Covalent division of SpringSource[2]. He contributed more than 10,000 commits to Tomcat. In a commit message, he left the complaint as follow:

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

In this example, we find that even experienced programmers can have problems in understanding their own code changes, when they maintain concurrent code. Our results can resolve such confusions.

Benefit 2. The results can be useful to improve existing tools. For example, Meng *et al.* [16] proposed an approach that applies changes systematically based on a given example. With extensions, it can be feasible to apply our extract change patterns to update concurrent code.

However, to fulfill the above benefits, we have to overcome the following challenges:

Challenge 1. To ensure the reliability of our result, we have to collect many code changes that are related to concurrent programming. It is tedious to manually collect many related code changes for analysis. Tian *et al.* [21] worked on a similar research problem. They proposed an approach that identifies bug fixes from commits. Their results show that even advanced techniques can fail to identify many desirable commits.

Challenge 2. The changes on concurrent code can be complicated. A recent study [22] showed that only 38% commits are compilable. To analyze code that is not compilable, researchers typically use partial-code tools such as PPA [6] and ChangeDistiller [8] to analyze commits. However, as partial programs lose information, partial-code tools are imprecise and typically do not support advanced analysis. Furthermore,

[1]http://tomcat.apache.org/whoweare.html
[2]https://sourceforge.net/projects/covalent/

as we do not know what patterns can be followed, it is difficult to implement an automatic tool. As a result, it is inevitable to take much human effort when we conduct the empirical study.

In this paper, we conduct an empirical study on 98,325 commits that are collected from six popular open-source projects. To reduce the effort of manual inspection, we implement a set of tools that collect and identify concurrency-related commits automatically (see Section II-C for details). With its support, in total, we identified 11,868 concurrency-related commits, and manually analyzed 696 such commits. Based on our results, this paper makes the following contributions:

- The first analysis on the change patterns of maintaining concurrent programs. Based on our results, we summarize five change patterns, and we present their examples for explanation. We find that following such change patterns, during software maintenance, programmers can modify concurrent code to repair bugs, improve performance, and change functions of their code. Furthermore, we find that maintaining concurrent code is not a one-direction migration. Due to various considerations, programmers can apply seemingly contradictory changes, and even revert their changes. Sometimes, programmers can even make changes, before they fully understand the consequences of their changes.
- An application of our change patterns in real code. In particular, we search the latest versions of three projects for chances to apply our change patterns, and synthesize three pull requests according to our change patterns. Two of our pull requests are already confirmed and accepted by their programmers. However, our results also reveal that it needs much experience and understanding to leverage our change patterns.

II. Methodology

A. Research questions

To understand how concurrent code is maintained, in this study, we focus on the following research questions:

RQ1. What change patterns are followed when programmers maintain concurrent code?

In this paper, we define change patterns as abstract description of similar code changes that appear many times in code revision history. In our study, we summarize concurrency-related commits into five change patterns, and present examples to explain our change patterns (Section III-A).

RQ2. How useful are our extracted change patterns, when programmers maintain concurrent code?

To assess the usefulness of our extracted change patterns, we manually apply our change patterns in three pull requests. Two of them are accepted by their programmers (Section III-B).

B. Dataset

In this study, we collected commits from six popular and active Apache projects. Table I shows the details of our data set. These projects cover various types of projects such as distributed computing, web server, database, information retrieval, and network. Column "#Commits" lists number of commits.

TABLE I: Selected Commits

Project	#Commits	#Concurrency	#Manual
Hadoop	14,930	2,739	64
Tomcat	17,731	1,963	207
Cassandra	21,982	1,904	78
Lucene-solr	26,152	2,375	99
Netty	7,759	1,387	210
Flink	9,771	1,500	38
Total	98,325	11,868	696

Column "#Concurrency" lists number of concurrency-related commits. From these concurrency-related commits, we selected a subset for manual analysis. Column "#Manual" lists number of our selected commits. We checked out all the commits in December 2016.

C. Study mechanism

As introduced in Section I, it is quite difficult to implement a single tool to automate our analysis. Instead, we employ and implement a set of tools to reduce the analysis effort. Inevitably, we have to introduce manual analysis in RQ1. Our study mechanism has the following steps:

1) Step 1. Collecting commits: All the projects in our study use Git as their version control system. We implement a tool to check out all their commits. A typical commit log contains a commit id, an author name, the commit date, and a message. Once we get a commit id, our tool uses the `git show` command to list details, and then uses the textual `diff` command to produce its change hunks.

2) Step 2. Identifying commits for the follow-up analysis: From collected commits, the second step is to extract commits that are related to concurrent code. Here, we consider that a commit is a *concurrency-related commit*, if the commit involves synchronization, thread, or concurrent API classes. A commit has a commit message that often explains which files are modified and why programmers make such modifications. Our tool builds queries to search for concurrency-related commits. The built queries contain concurrency-related keywords. We choose 96 keywords such as `synchronized`, `volatile`, and concurrent API class names. The full list can be found in our project homepage: https://github.com/qwordy/Research. However, this selector selects 11,868 commits that are too many for manual analysis. We selected a subset from them for manual analysis by checking whether a commit message contains concurrency-related keywords. The first selector selects 11,868 commits from 98,325 commits. The second selector selects 561 commits from the 11,868 commits that are output of the first selector. The size of the final selected set is 561. The precision is 0.67 based on a manual inspection.

The textual matching method can lose some useful commits. We use a machine learning method to select concurrency-related commits from the 11,868 commits as supplements. Researchers have explored related problems. For example, Tian *et al.* [21] proposed an approach that identifies bug fixing patches with classification techniques. Motivated by their approach, we train a classifier to predict concurrency-related commits. Our tool analyzes change hunks that are produced by the `diff` command, and extracts code features.

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

As shown in Table II, our tool extracts 12 features from each commit. The first column shows feature names, and the second column shows explanations. The keywords are the same as the concurrency-related keywords used in the previous paragragh. Our tool employs the SVM [5] algorithm to identify concurrency-related commits. In particular, we use LIBSVM [4]. We randomly select 48 commits as a training set. We build features of them and label them. The 48 commits have 15 positive instances and 35 negative ones. Then we train a model and use it to classify commits. It selects 135 positive commits. The precision is 0.74 based on a manual inspection. The accurate recall is unavailable because our goal is to predict these positive instances. We selected 696 commits for manual analysis in total.

3) Step 3. Analyzing commits according to different research questions: We then conduct detailed analysis according to our research questions.

RQ1. Determining change patterns. To explore this research question, we analyzed each selected commit for their change patterns. For each commit, we first read the message and the corresponding issue to understand why programmers make the commit. After that, we scan change hunks to understand the details. Here, we classify concurrency-related commits into different categories, mainly according to our observed code changes such as the modifications on code elements, parallel libraries, and control flows.

RQ2. Exploring the usefulness of our change patterns. We prepare a set of keywords for each change pattern, and search Github for code where the pattern can apply. For example, we use `synchronized`, `put` or `get` as keywords to search code pieces that manually handle synchronization of collections. We find numerous code pieces in the search results. We manually check the code and decide whether our patterns apply on such code. If it is, we fork the project; make our changes; and submit our pull request.

III. Results

A. RQ1. Change patterns

Table III shows an overview of our extracted change patterns. Each row is an example of a certain change pattern. The first column is the sequence number of examples. The "Source Code" column shows the concrete source code of examples. The left is the original code and the right is the modified code. We align the corresponding statements. We use different colors to mark modified lines. The "Simplified Code" column shows the simplified code of the source code. They are short as they ignore the specific statements.

1. Changing lock types. It is feasible to lock resources with different mechanisms. For example, Java has a keyword, `synchronized`. The keyword can lock a block of code lines. With the keyword, programmers do not have to acquire and release resources explicitly. Alternatively, programmers can explicitly lock resources with APIs (*e.g.*, `ReentrantLock`). Explicit locks offer more features than the `synchronized` keyword does. As another example, besides exclusive locks, programmers can use shared locks, that allow multiple threads to hold the lock in certain conditions.

We find that programmers can replace the `synchronized` keyword with parallel API classes. For example, the first item of Table III comes from YARN-5825[3]. To improve the performance, programmers replaced the `synchronized` with the `getReadLock` method. The method returns a shared lock, so multiple threads can read the query simultaneously.

Meanwhile, we find that programmers can replace parallel API classes with the `synchronized` keyword. For example, the second item of Table III comes from an unreported Tomcat bug. We find it through our SVM classifier.

```
1  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.
```

As the above message explains, a developer complained that the `ReadWriteLock` method does not guard the `WeakHashMap` variable, since the `get()` method can modify the `WeakHashMap` variable, and the modification can bypass the lock. In this example, programmers fixed the problem by replacing the methods with the `synchronized` keyword.

2. Changing locked variables. A program needs to lock variables before it enters critical section bodies. During software maintenance, programmers can change locked variables, and we find that the main purpose is to repair bugs. For example, the third item of Table III comes from FLINK-1419. This bug complains that `DistributedCache` does not preserve files for subsequent operations. Based on its discussions, we understand that in the buggy file, programmers lock `count`, while they shall lock `lock`. To fully fix the bug, programmers also modified the critical sections and the `finally` clause.

As another example, when repairing bugs, programmers can add new locks. For example, the fourth item of Table III comes from Tomcat[4]. It includes the following message:

```
1  Reported by RV-Predict (a dynamic race detector) when
       running the test suite: Data race on ...)
```

The data race indicates that the `isAccessed()` is not locked, so programmers add the `synchronized` keyword to allow locking on the method.

[3]https://issues.apache.org/jira/browse/YARN-5825 The URLs of Apache issues can be built by replacing the above URL with their issue number.

[4]https://bz.apache.org/bugzilla/show_bug.cgi?id=58386 The URLs of Tomcat issues can be built by replacing the above URL with their issue number.

TABLE III: Change patterns

#	Source Code		Simplified Code	
	Original	Modified	Original	Modified

1	LeafQueue leafQueue = ...;
	-synchronized (leafQueue) {
	  57 LOC
	}
		LeafQueue leafQueue = ...;
		+try { leafQueue.getReadLock().lock();
		  57 LOC
		+} finally {
		+ leafQueue.getReadLock().unlock();}
			synchronized (obj)
			{
			  ...
			}
				try {obj.lock();
				  ...
				} finally {
				  obj.unlock();
				}

2	-Lock readlock =
	- classLoaderContainerMapLock.readLock();
	-try { readlock.lock();
	- result = classLoaderContainerMap.get(tccl);
	-} finally {readlock.unlock();}
	-if (result == null) { Lock writelock =
	- classLoaderContainerMapLock.writeLock();
	- try { writeLock.lock();
	  result = classLoaderContainerMap.get(tccl);
	  if (result == null) {
	  result = new ServerContainerImpl();
	  classLoaderContainerMap.put(tccl,result);}
	- } finally {writeLock.unlock();}
		+synchronized (classLoaderContainerMapLock) {
		
		  result = classLoaderContainerMap.get(tccl);
		  if (result == null) {
		  result = new ServerContainerImpl();
		  classLoaderContainerMap.put(tccl,result);}
		}
			try {
			  readLock.lock();
			  read operations
			} finally {
			  readLock.unlock();
			}
			try {
			  writeLock.lock();
			  write operations
			} finally {
			 writeLock.unlock();
			}
				synchronized {
				  all operations
				}

3	static final Object lock = new Object();
	Map<...> count = new HashMap<>();
	-synchronized (count) {
	- Pair<Job, String> key =
	- new ImmutablePair<>(jobID, name);
	
	- if (count.containsKey(key)) {
	- count.put(key, count.get(key) + 1);
	- } else {count.put(key, 1);}}
		static final Object lock = new Object();
		Map<...> count = new HashMap<>();
		+synchronized(lock)
		+ if (!jobCounts.containsKey(jobID)) {
		+ jobCounts.put(jobID, new HashMap<>());}
		+ Map<...> count = jobCounts.get(jobID);
		+ if (count.containsKey(name)) {
		+ count.put(name, count.get(name) + 1);
		+ } else {count.put(name, 1);}}
			synchronized (obj1)
			{
			  ...
			}
				synchronized
				(obj2) {
				  ...
				}

4	-public boolean isAccessed() {
	  return this.accessed;}
		+public synchronized boolean isAccessed() {
		  return this.accessed;}
			void foo() {...}	synchronized void
				foo() {...}

5	synchronized (buffers) { if (...) {
	
	- if (spillWriter != null) {
	- spillWriter.close();}
	  isFinished = true;}}
		synchronized (buffers) { if (...) {
		  isFinished = true;}}
		+if (spillWriter != null) {
		+ spillWriter.close();}
			synchronized(obj)
			{
			  statements1
			  statements2 }
				synchronized(obj)
				{
				  statements2 }
				Statements1

6	-synchronized void reset() {
	
	  map.clear();
	  members = EMPTY_MEMBERS;}
		+final Object membersLock = new Object();
		+void reset() { synchronized (membersLock) {
		  map.clear();
		  members = EMPTY_MEMBERS;}}
			synchronized void
			foo() {
			  ... }
				void foo() {
				  synchronized
				(obj) {
				  ... }}

7	-synchronized void enqueue(final long seqno,
	  final boolean lastPacketInBlock,
	  final long offsetInBlock) {
	- if (running) {
	  final Packet p = new Packet(...);
	  LOG.debug(...);
	
	  ackQueue.addLast(p); notifyAll();}}
		+void enqueue(final long seqno,
		  final boolean lastPacketInBlock,
		  final long offsetInBlock) {
		
		  final Packet p = new Packet(...);
		  LOG.debug(...);
		+ synchronized (this) { if (running) {
		  ackQueue.addLast(p); notifyAll();}}}
			synchronized void
			foo(...) {
			  statements1
			  statements2
			}
				statements1
				synchronized (obj)
				{
				  statements2
				}

8	-Membership membership = null;
	public boolean hasMembers() {
	  if (membership == null) setupMembership();
	  return membership.hasMembers();}
	synchronized void setupMembership() {
	  if (membership == null) {
	  membership = new Membership(...);}}
		+volatile Membership membership = null;
		public boolean hasMembers() {
		  if (membership == null) setupMembership();
		  return membership.hasMembers();}
		synchronized void setupMembership() {
		  if (membership == null) {
		  membership = new Membership(...);}}
			T foo;	volatile T foo;

9	-volatile int requestCount; - requestCount++;	+final AtomicInteger requestCount = + new AtomicInteger(0); + requestCount.incrementAndGet();	volatile T foo;	TT foo;

In some other cases, we find that programmers can refine their locked resources to improve performance. For example, the sixth item of Table III comes from Tomcat 58382. The original code locks the instance of a class, but the modified code locks only the `membersLock` field.

3. Modifications inside critical section bodies. A critical section is a code block that is executed, when a thread locks the corresponding resources. We notice that even modifications inside critical section bodies can repair concurrency bugs. For example, the fifth item of Table III comes from FLINK-2384. It is caused by an implicit lock in the `spillWriter.close()` method. As programmers typically do not know such locks inside APIs, the lock leads to the deadlock. Indeed, we find that a recent benchmark [11] includes a similar concurrency bug, and Lin *et al.* [12] proposed an approach that detects such implicit locks inside APIs. In this example, programmers move the `spillWriter.close()` method outside the critical section body to resolve the deadlock.

Besides the above example, the majority of modifications on critical section bodies indicates new functionalities or refactoring. For example, the seventh item of Table III comes from HDFS-4200. To reduce the size of a critical section body, programmers refactor the body into several methods. Indeed, this issue involves modifications that are related to even more change patterns (*e.g.*, adding locked variables).

4. Changing the `volatile` keyword. In Java, the `volatile` keyword denotes a variable that must be accessed from main memory and disables reordering. Although it can improve the overall performance, races can occur, when multiple threads read and write `volatile` variables simultaneously.

We find that programmers can add the `volatile` keyword to improve performance. For example, the eighth item of Table III comes from Tomcat 58392. It reports a data race, and the bug was repaired by adding the `volatile` keyword.

Besides adding the keyword, we find that programmers have to remove the `volatile` keyword, since they do not fully understand its meanings. For example, the ninth item of Table III comes from a Tomcat bug. In particular, programmers auto-increment a `volatile` variable, but such an action is not atomic. In this fix, programmers have to remove the keyword.

5. Replacing self-written code with Parallel APIs. Programmers can implement concurrent code by themselves, but later they realize that it is easier to call APIs that already implement their functionalities. For example, a previous version of Hadoop has the following code:

```
1  private volatile long genstamp;
2  public synchronized long nextStamp() {
3    this.genstamp++;
4    return this.genstamp; }
```

Later, a programmer realized that it is better to replace the above code with the `AtomicLong` class. He reported a major issue (HDFS-4029). Here, `AtomicLong` is a thread-safe version of type long. It allows updating a `Long` value without explicit synchronization and it is fast.

```
1  private volatile long genstamp;
2  public long nextStamp() {
3    return genstamp.incrementAndGet(); }
```

Besides replacing directly, programmers can replace their code with parallel APIs that implement similar functions. For example, the below code comes from LUCENE-2779:

```
1  protected ... fileMap = new HashMap<...>();
2  public final boolean fileExists(String name) {
3    ensureOpen();
4    RAMFile file;
5    synchronized (this) { file = fileMap.get(name); }
6    return file != null; }
```

Instead of handling synchronization by themselves, programmers replaced the above code with a parallel API:

```
1  protected ... fileMap = new ConcurrentHashMap<...>();
2  public final boolean fileExists(String name) {
3    ensureOpen();
4    return fileMap.containsKey(name); }
```

In summary, we identify five types of change patterns in total. The percentages of their occurrence are 4%, 3.5%, 29%, 34.5% and 29% respectively. First, we find that programmers can modify parallel keywords and locked variables, and such modifications are mainly for repairing bugs and improving performance. Second, we notice modifications on critical section bodies, and such modifications often indicate new functions. Finally, we notice that self-written code is replaced with corresponding parallel APIs. In most cases, we find that maintaining concurrent code is not a one-direction migration.

B. RQ2. The usefulness of our change patterns.

Once developers know and understand these change patterns well in concurrent programming, they can refer to these change patterns to see whether one of them matches the condition when they are writing or maintaining concurrent code and then make the change. However, it is impractical for us to read vast concurrent code and find examples to apply the change patterns. Instead, we use a keyword search method to find possible code context to apply the change patterns.

In total, we made three pull requests. We made the first pull request on Schmince-2. The original code is as follow:

```
1  public class DRandom {
2    private ... random = new ThreadLocal<Random>() {
3      protected Random initialValue() {return new Random
           ();}};
4    public static Random get() {return random.get();}}
```

The above code implements a class that generates random values for multiple threads. We find that J2SE provides the `ThreadLocalRandom` class that implements the identical function. According to our fifth change pattern, we made a pull request to replace the above code with the corresponding API. This pull request is already confirmed.

We made the second pull request on UnifiedEmail. It has the following code:

```
1  static NMap map = null;
2  static synchronized NMap getNMap(Context context) {
3    if (map == null) {
4      map = new NMap(); ...; }
5    return map; }
```

If the `map` field is `null`, the above method creates a new map and assigns the new map to the field. To allow multiple threads to call the methods, programmers add the `synchronized` keyword to the method. We believe that when `map` is not `null`, the lock is unnecessary, since it does not change the field. According to our second change pattern, we made the following modification to synchronize only the lines that modify the field:

```
1  static volatile NMap map = null;
2  static NMap getNMap(Context context) {
3    if (map == null) {
4      synchronized (NotificationUtils.class) {
5        if (map == null) {
6          map = new NMap(); ...; }}}
7    return map; }
```

The owner of the project deleted our pull request. We checked the pull requests of the project and found that there is no open or closed pull request. This project has more than 19,000 commits. It is possible that it has a strict policy of introducing external source code.

We made the third pull request on Spider4java. It is a Java web crawler. It has the following code:

```
1  public class Counter {
2    protected int count;
3    public Counter() { count = 0; }
4    public synchronized void increment() {count=count+1;}
5    public synchronized int getValue() {return count;}}
```

The above class implements a counter for multiple threads. As J2SE provides the identical API, `AtomicInteger`. Our pull request has been accepted, and its code is as follow:

```
1  public class Counter {
2    protected AtomicInteger count;
3    public Counter() { count = new AtomicInteger(); }
4    public void increment() { count.getAndIncrement(); }
5    public int getValue() { return count.get(); }}
```

In summary, our results show that our change patterns are repetitive in future maintenance, and programmers confirmed that our change patterns are useful. However, our results also reveal that it needs much programming experience to fully unleash the potential of our change patterns.

C. Threats to Validity

The threats to internal validity include that our tool can omit some concurrency commits, although we have used both the query-based search and a classifier in our study. We did not conduct a thorough evaluation on our classifier. The threat could be reduced by more advanced identification techniques. The threats to internal validity also include obsolete commits. These commits may present obsolete or even wrong usages. To reduce the threat, in our study, we prefer to recent commits. The threats to external validity include our selected projects and programming language. The number of the projects we select is small. They are all Java-based Apache projects. The threat could be reduced by introducing more projects and languages in future work.

IV. Related work

Empirical studies on concurrent programming. In literature, researchers have conducted various empirical studies to understand concurrent programming. Pinto *et al.* [18] conducted a large scale study on the usage of concurrency in Java, and Wu *et al.* [23] replicated their study with C++. Okur and Dig [17] studied how developers use parallel libraries in C#. David *et al.* [7] conducted an empirical study to investigate synchronization at both hardware and software levels. Sadowski *et al.* [19] studied the evolution of data races, and they found that many data races always exist. Xin *et al.* [25] conducted an empirical study on lock usage, and they found that most functions acquire only a lock. Lu *et al.* [13] studied characteristics of real world concurrency bugs. The above approaches do not analyze change patterns, which are complemented by our study.

Identification of commits. Zhong and Su [26] relied on simple heuristic to identify bug fixes from commits. Tian *et al.* [21] trained a classifier to identify bug fixes based on their extracted features. Wu *et al.* [24] built the links between bug fixes and their reports based their similarity values. The above approaches focus on identifing bug fixes from commits. In our study, we implement a tool that identifies concurrency-related commits, complementing the above approaches.

V. Conclusion

Concurrent programming is challenging, and a mistake can introduce hidden bugs that are difficult to be detected. During software maintenance, programmers have to handle concurrent code carefully. Researchers have conducted various empirical studies to understand concurrent programming. However, how programmers maintain concurrent code is still rarely studied. In this paper, we conduct an empirical study to understand the change patterns and other perspectives of concurrent programming. Based on our analysis results, we summarize five change patterns. We show that such change patterns are repetitive in future maintenance, and programmers have confirmed the usefulness of our extracted patterns.

References

[1] Y. Ahn, J. Suh, S. Kim, and H. Kim. The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):71–85, 2003.

[2] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. In *ESEC/FSE*, pages 385–394, 2007.

[3] H. Borges. On the popularity of github software. In *ICSME*, page 618, 2016.

[4] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[5] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[6] B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *OOPSLA*, pages 313–328, 2008.

[7] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48, 2013.

[8] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Transactions on Software Engineering*, 33(11):725–743, 2007.

[9] D. M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[10] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu. What change history tells us about thread synchronization. In *ESEC/FSE*, pages 426–438, 2015.

[11] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao. JaConTeBe: A benchmark suite of real-world Java concurrency bugs. In *ASE*, pages 178–198, 2015.

[12] Z. Lin, H. Zhong, Y. Chen, and J. Zhao. LockPeeker: detecting latent locks in Java APIs. In *ASE*, pages 368–378, 2016.

[13] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[14] T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the android ecosystem. In *ICSM*, pages 70–79, 2013.

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

[16] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[17] S. Okur and D. Dig. How do developers use parallel libraries? In *FSE*, page 54, 2012.

[18] 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*, 106:59–81, 2015.

[19] C. Sadowski, J. Yi, and S. Kim. The evolution of data races. In *MSR*, pages 171–174, 2012.

[20] G. Santos, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente. System specific, source code transformations. In *ICSME*, pages 221–230, 2015.

[21] Y. Tian, J. Lawall, and D. Lo. Identifying Linux bug fixing patches. In *ICSE*, pages 386–396, 2012.

[22] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 2016.

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

[24] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *ESEC/FSE*, pages 15–25, 2011.

[25] 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*, pages 100–109, 2013.

[26] H. Zhong and Z. Su. An empirical study on real bug fixes. In *ICSE*, pages 913–923, 2015.