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 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 programs are known hard to write correctly for multiple threads accessing objects simultaneously or depending on each other usually need complex synchronization and hard to debug for the uncertainty of thread interleaving which makes it difficult to reproduce the bug. Developers often struggle with various of synchronization methods and subtle concurrent bugs. There are much research about concurrent programming in the literature such as data race detection, atomicity violation detection or deadlock detection. Some empirical work also give us much help. Rui Gu [1] studied change history of thread synchronization. Gustavo Pinto [2] did a large-scale study on the usage of Javas concurrent programming constructs.

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. 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 studies focus on topics of software evolution such as refactoring, transformation patterns. Gustavo Santos [3] studied system specific, source code transformations.

We studied concurrent programs from a perspective of software evolution history and found many change patterns about concurrent programming.

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:

1.

2. We identify and classify change patterns in concurrent code and observe some interesting findings.

3. 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 the project sources of our study, research questions and methods of doing the study. We have developed a tool supporting the empirical study.

A. Data set

We investigate 8 Java open-source projects from Github including Hadoop, Tomcat, Cassandra, Lucene-solr, Netty, Flink, Guava 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 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

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

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 1 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 better, we proposed 4 research questions:

RQ1. How many change patterns in concurrent programming?

Some change patterns are system-specific while some are global, which can be considered as knowledge. Developers made numerous commits to the project repository during software's whole life. Researchers have found some changes are similar , known as change patterns. We have a belief that there are also many change patterns in concurrent programming.

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

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. Java programming language provides convenient built-in concurrent libraries and users can also invoke third-party libraries. Although developers can use their own concurrent related classes or third-party libraries, they are always using the facilities provided by JDK by default. We want to know how frequent do concurrent related code modification 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 always are 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 possible reasons hidden behind the phenomenon.

RQ4. How can these change patterns in history guide the development?

In order to better demonstrate these change patterns can really help developers understand concurrent programming practice, we are going to find the appropriate context in open-source projects and pull requests of applying the change patterns.

C. Tool support

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

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[4]. 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.

We use the SVM[5] 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[6]. 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 change patterns in concurrent programming?

There are so many different concurrent related changes in the code history. The change patterns in concurrent programming can be divided into two categories: (1) Class replacement; (2) Adjustment of concurrency control.

Example 1

```
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.

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

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

D. RQ4. How can these change patterns in history guide the future development?

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.

(2)

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.

(3) Researchers

V. Related work

Studies on concurrent program

Program transformation

VI. Conclusion

We conduct a study on change patterns in concurrent programming.

Acknowledgment

The authors would like to thank...

References

[1] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu, "What change history tells us about thread synchronization," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 426–438. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786815

[2] 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. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2015.04.064

[3] G. Santos, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "System specific, source code transformations," in *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, 2015, pp. 221–230. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332468

[4] Y. Tian, J. L. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 386–396. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227176

[5] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: http://dx.doi.org/10.1007/BF00994018

[6] 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.