การพัฒนาอัลกอริทึมที่ใช้ในการตรวจจับการติดตาย
สำหรับการระบุการติดตายที่มีแนวโน้มที่จะเกิดขึ้น

นางสาวสุวารินทร์ พลอยศรี

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมซอฟต์แวร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2557

A DEVELOPMENT OF THE DEADLOCK DETECTION ALGORITHM FOR IDENTIFYING POTENTIAL DEADLOCKS

Miss Suvarin Ploysri

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science Program in Software Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2014

Thesis Title	A DEVELOPMENT OF THE DEADLOCK DETECTION ALGORITHM FOR IDENTIFYING POTENTIAL DEADLOCKS
By	Miss Suvarin Ploysri
Field of Study	Software Engineering
Thesis Advisor	Associate Professor Wanchai Rivepiboon, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

....Dean of the Faculty of Engineering
(Professor Bundhit Eua-arporn, Ph.D.)

THESIS COMMITTEE

....Chairman
(Associate Professor Wiwat Vatanawood, Ph.D.)

....Thesis Advisor
(Associate Professor Wanchai Rivepiboon, Ph.D.)

....Examiner
(Associate Professor Twittie Senivongse, Ph.D.)

....External Examiner
(Associate Professor Somchai Prakancharoen, Ph.D.)

สุวารินทร์ พลอยศรี : การพัฒนาอัลกอริทึมที่ใช้ในการตรวจจับการติดตายสำหรับการระบุการติดตายที่มีแนวโน้มที่จะเกิดขึ้น (A DEVELOPMENT OF THE DEADLOCK DETECTION ALGORITHM FOR IDENTIFYING POTENTIAL DEADLOCKS) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: รศ. ดร. วันชัย ริ้วไพบูลย์, 198 หน้า.

ในปัจจุบันการพัฒนาอินเตอร์เฟสของชุดคำสั่งของโปรแกรมประยุกต์แบบมัลติเธรด(เอพีไอ) เพื่อการใช้งานเฉพาะอย่างนั้น มีการพัฒนาอย่างกว้างขวางมาก อย่างไรก็ตามปัญหาที่เกิดจากเอพีไอแบบมัลติเธรดนั้นก็ได้เกิดขึ้นมาด้วย นั่นก็คือการติดตาย การติดตายในเอพีไอแบบมัลติเธรดเป็นสิ่งที่น่ากังวลมากที่สุดเพราะว่า เราไม่สามารถค้นหาการติดตายในโปรแกรมประยุกต์และค่าใช้จ่ายสำหรับการแก้ไขข้อบกพร่องภายหลังเฟสการพัฒนาจะสูงมากขึ้นและปัญหามีความซับซ้อนมากขึ้น การตรวจจับการติดตายในเอพีไอแบบมัลติเธรดในเฟสต้นของวงจรการพัฒนาซอฟต์แวร์หรือการใช้การวิเคราะห์แบบสถิตย์คงเป็นวิธีที่มีประสิทธิภาพมากกว่าวิธีอื่น เนื่องจากเรายังไม่ทราบว่าผู้เขียนโปรแกรมจะนำเอาเอพีไอแบบมัลติเธรดไปใช้อย่างไร การตรวจจับทุกๆ ส่วนของชุดคำสั่งจึงถูกนำมาใช้ในงานวิจัยฉบับนี้ เราประสบความสำเร็จในการพัฒนาอัลกอริทึมที่ใช้ในการตรวจจับการติดตาย โดยนำแนวคิด ของมายูร์ เนก ซึ่งกล่าวไว้ในปี 2009 ว่า เงื่อนไขของการเกิดการติดตายนั้นมี 5 ประเภทและ แนวคิดของแฟรงก์ อ็อตโต ซึ่งกล่าวไว้ในปี 2008 ว่า รูปแบบของรหัสคำสั่งนั้นมี 2 ประเภท ในงานวิจัยชิ้นนี้ได้เสนอ 7 เงื่อนไขของการเกิดการติดตาย ซึ่งได้แก่ เงื่อนไขสมนาม เงื่อนไขการล็อคด้วยลำดับผันกลับหรือเงื่อนไขล็อคพึ่งพาแบบวงกลม เงื่อนไขหลบหนี เงื่อนไขแบบขนาน เงื่อนไขไม่มีล็อคป้องกัน และเงื่อนไขล็อคฟุ่มเฟือย มาใช้ในการพัฒนาอัลกอรีทึมที่ใช้ในการตรวจจับการติดตาย และนอกจากนี้แล้ว เราได้มีการพัฒนาเครื่องมือต้นแบบที่ใช้ในการการตรวจจับการติดตายเพื่อแสดงให้เห็นถึงการใช้งานอัลกอริทึมที่ใช้ในการตรวจจับการติดตาย ผลของการพัฒนาอัลกอริทึมที่ใช้ในการตรวจจับการติดตายและเครื่องมือที่ใช้ในการตรวจจับการติดตายนั้น ให้ผลเป็นที่น่าพอใจ และ ถูกต้องตามที่คาดไว้

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต	....
สาขาวิชา	วิศวกรรมซอฟต์แวร์	ลายมือชื่อ อ.ที่ปรึกษาหลัก	....
ปีการศึกษา	2557		

5471023921 : MAJOR SOFTWARE ENGINEERING

KEYWORDS: DEADLOCK / DEADLOCK DETECTION ALGORITHM / STATIC ANALYSIS / MULTITHREADING / JAVA / API

SUVARIN PLOYSRI: A DEVELOPMENT OF THE DEADLOCK DETECTION ALGORITHM FOR IDENTIFYING POTENTIAL DEADLOCKS. ADVISOR: ASSOC. PROF. WANCHAI RIVEPIBOON, Ph.D., 198 pp.

Currently, developing a multithreading Application Programming Interface (API) for special use is extensive. Deadlock in the multithreading API is the most concerned problem because we cannot find the deadlock in the application and the cost for defect fixing later on the development phase is even higher and more complex. Detecting deadlock in early phase of the Software Development Life Cycle or using static analysis is the way more effective. Since we still do not know how developer uses the API, detecting deadlock in the source code using static analysis is our selection for this research. We have successfully developed the Deadlock Detection Algorithm that brings the concept of five deadlock conditions by Mayur Naik published in 2009 and two code patterns by Frank Otto published in 2008. We present seven deadlock conditions that are the Aliasing Condition, the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition, the Escaping Condition, the Parallel Condition, the Non-Guarded Lock Condition and the Superfluous Lock Condition. In addition, we develop the prototype of the Deadlock Detection Tool to demonstrate the use of the Deadlock Detection Algorithm. The result of the implementation of the Deadlock Detection Algorithm and the Deadlock Detection Tool are satisfied and provide correct result as expected.

Department:	Computer Engineering	Student's Signature	....
Field of Study:	Software Engineering	Advisor's Signature	....
Academic Year:	2014		

ACKNOWLEDGEMENTS

First and foremost I wish to express my deepest gratitude to my thesis advisor, Associate Professor Dr. Wanchai Rivepiboon, for his continuous support in the entire of my thesis and life. Without his encouragement, understanding, caring and excellent guidance, I could not have finished this thesis. I would like to thank the rest of my thesis committee, Associate Professor Dr. Wiwat Vatanawood, Associate Professor Dr. Twittie Senivongse and Associate Professor Dr. Somchai Prakancharoen, who gave me comments and reviewed my work. In addition, I offer my sincerest gratitude to the faculty and officers of Computer Engineering and Chulalongkorn University.

Besides my academic, I would like to thank for Thomson Reuters (Software) Thailand for sponsorship, support and opportunity to develop my knowledge and ability. In addition, I would like to thank for my ex-colleagues in Thomson Reuters, especially, TRDC support team.

I would like to thank for my all of my classmates, especially my 10th group for help, support, encouragement, sincere and friendship.

Above all, I am deeply grateful for my family, relatives and close friends that always stay by my side through the good and bad and pray for me throughout the time of my study. This thesis is heartily dedicated to my mother and father who value education and goodness.

CONTENTS

Page

Page

Page

LIST OF FIGURES

LIST OF TABLES

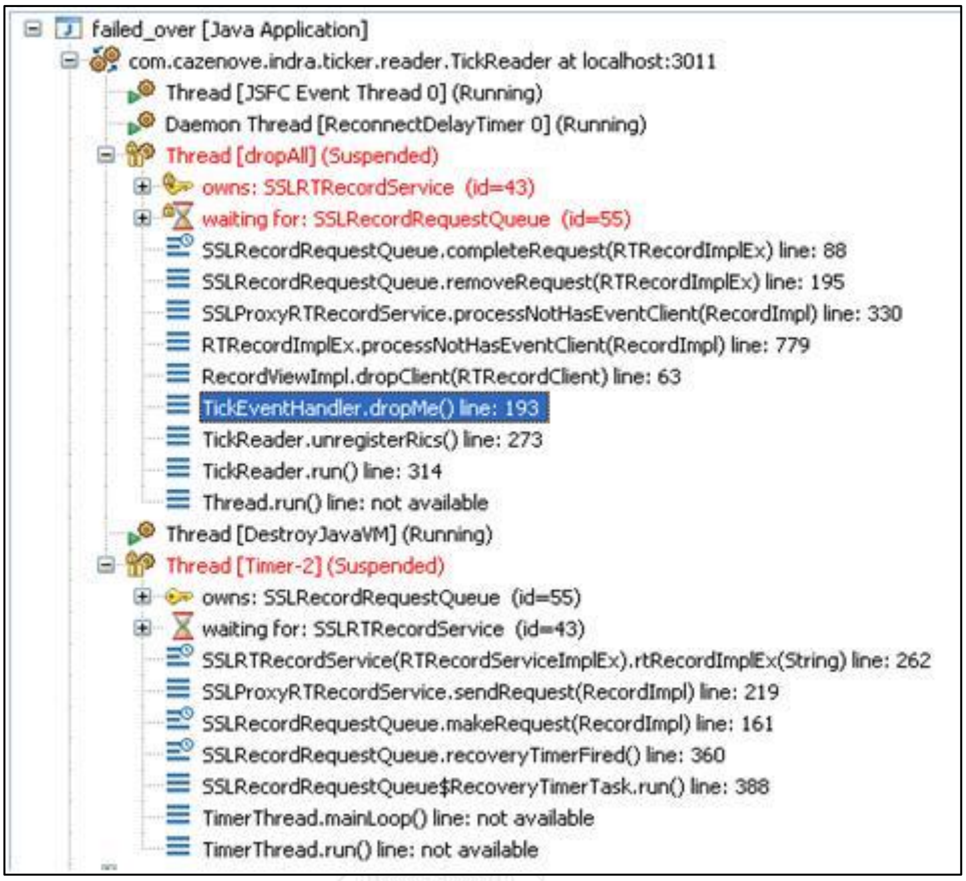
CHAPTER I
INTRODUCTION

This chapter provides an introduction of this thesis; including the motivation, objective, scope, contribution, publications, research methodology and the organization of this Thesis.

1.1. Motivation

In this present, the multithreading API developing for specific usage is prevalence. The multi-core technology leads developing software design [1]. Therefore the application is able to work faster and get more result than a sequential application in a time. However, there is a trade-off that the application potentially encounters the deadlock problem [2]. And what is a "Deadlock"? Mayur Naik, David Gay, Change-Seo Park and Koushik Sen also presumed that "Deadlock is an unintended condition in which a set of threads blocks forever because each thread in the set is waiting to acquire a lock already held by another thread in the set" [3].

In support and consulting service for a multithreading API, the API defects deadlocks and the problems occur after applications are developed or released. When the application calls the API that is underlying layer of the application, the API will create several threads and there are more than one thread lock and wait for the same objects. Therefore when the application calls the API that is defect with deadlocks, the application encounters deadlock problems. In problem investigation work for the deadlock in the API support and consulting service, a customer provides the stack trace to report deadlock issue to the support consultant. The support consultant has to analyze the stack trace that relates to the API behavior to reproduce the problem. They have to mark the break points reported in the stack trace showing in Figure 1 and then debug into the source code of the API to find the root cause of the deadlock.



```
failed_over [Java Application]
  com.cazenove.indra.ticker.reader.TickReader at localhost:3011
    Thread [JSFC Event Thread 0] (Running)
    Daemon Thread [ReconnectDelayTimer 0] (Running)
    Thread [dropAll] (Suspended)
      owns: SSLRTRecordService  (id=43)
      waiting for: SSLRecordRequestQueue  (id=55)
      SSLRecordRequestQueue.completeRequest(RTRecordImplEx) line: 88
      SSLRecordRequestQueue.removeRequest(RTRecordImplEx) line: 195
      SSLProxyRTRecordService.processNotHasEventClient(RecordImpl) line: 330
      RTRecordImplEx.processNotHasEventClient(RecordImpl) line: 779
      RecordViewImpl.dropClient(RTRecordClient) line: 63
      TickEventHandler.dropMe() line: 193
      TickReader.unregisterRics() line: 273
      TickReader.run() line: 314
      Thread.run() line: not available
    Thread [DestroyJavaVM] (Running)
    Thread [Timer-2] (Suspended)
      owns: SSLRecordRequestQueue  (id=55)
      waiting for: SSLRTRecordService  (id=43)
      SSLRTRecordService(RTRecordServiceImplEx).rtRecordImplEx(String) line: 262
      SSLProxyRTRecordService.sendRequest(RecordImpl) line: 219
      SSLRecordRequestQueue.makeRequest(RecordImpl) line: 161
      SSLRecordRequestQueue.recoveryTimerFired() line: 360
      SSLRecordRequestQueue$RecoveryTimerTask.run() line: 388
      TimerThread.mainLoop() line: not available
      TimerThread.run() line: not available
```

Figure 1 The stack trace of deadlock reported by a customer

The general root cause of the deadlock problem is incorrect synchronization of pair or more objects [4], incorrect ordering of lock acquisitions [2] or incorrect usage of the API. All these causes occur in the API that deadlock was not prevented (non-thread safe), miss-detected in the software development process or not tested enough.

The deadlock occurs in the application after the API was released to the customers that used the API to develop their applications. Actually, the deadlock should be found in the early phase of the Software Development Life Cycle of the API that is the Software Development phase. It can be also found in the Software Testing phase of the API. Hidden bugs in the API directly lead to increasing of development costs that are time and effort for investigating, fixing bugs and testing for the customers' application layer [5] to the API layer. The slower deadlock is found, the more the cost of the application increases. In the Design phase of the API we also conduct for deadlock prevention. However, the multithreading concept is error prone and difficult [6] therefore deadlock seems to be an inevitable problem. Lack of focusing on deadlock detection in the Software Testing process for concurrence programming of the API to save the cost should be disaster as well. And it causes the deadlock hidden, unexploded and not fixed until the package is released to the customer and the API is developed as an application on customer's sites. The deadlock prevention in the early phase of the Software Development can also help to get rid of deadlock. Knowing the structure and design of the software is also very important for deadlock detection and prevention in the Software Development phase because the developers can specify the pair or multiple objects or processes of incorrect synchronization statements from diagrams that are products of the Design phase. In addition, the behavior of programmers and incorrect coding can also lead to deadlock such as not emphasize the Software Design or copying-pasting code without awareness of incorrect usage of code or not use deadlock detection tools [6]. Lack of all these things causes deadlock in the API.

There are several researches concerning deadlock prevention suggested to detect the deadlock and find its cause early in the Software Design, the Software Development or the Software Testing. It is important to solve the error earlier to not to accumulate deadlock from the API to the applications.

The concept of developing the deadlock detection algorithm comes from processes in the Software Development Life Cycle that can encounter the deadlock. To detect the deadlock involves all Software process concepts to develop the deadlock detection algorithm. For deadlock prevention, in the Software Development phase, the developers use the object diagrams that are outputs from the Software Design phase to develop the API and know which objects and methods should be synchronized or not. In addition, there should be a flow diagram to present the API behavior when threads work together. In the Software Testing phase, the tester designs the path coverage test case to test the multithreading API. Using the concept of each process, the deadlock detection algorithm should be able to generate object and flow diagrams to elicit all paths of the multithreading API. In addition, detecting which synchronized objects, methods; including wait-notify methods and threads are in the multithreading API source code that cause deadlocks. The algorithm of the deadlock detection for the multithreading API should be the static analysis because it is more direct to detect the deadlock in the source code of the API than execute the application that is implemented on top of the API. In the other hands, the dynamic analysis for deadlock detection algorithm does not meet approach because the deadlock does not easily occur and reproduce at runtime, not cover all path of code, and there is no prior information why the deadlock occurs [2]. Therefore static analysis should be more suitable for testing, debugging [3] and developing the deadlock detection algorithm for the multithreading API.

1.2. Objective

The objective for this thesis is to research for the algorithm that is able to detect the potential deadlock using static analysis for the multithreading API developed by Java programming language. Moreover, use the algorithm to develop the prototype tool for deadlock detection.

1.3. Scope

Here below is the research scope of this thesis.

1. Detecting the deadlock in the multithreading API developed by Java Programming language using static analysis.
2. Detecting the deadlock that causes of using synchronized statements and wait-notify methods in the multithreading API developed by Java programming language.
3. Focusing on correctness of result after using the deadlock detection tool. In this research, it does not include the performance qualification for the Deadlock Detection Tool.
4. Researching on the application that uses the single thread to call to the multithreading API in order to not to affect the behavior of the multithreading API.
5. Resolving the deadlock issue is not a focus of this thesis.

1.4. Contribution

The outcomes of this research are the followings:

1. Getting defects early in the Software Development process, Software Testing process and Maintenance phase.
2. Using the result as related information to find the root cause of deadlock and solve the problem in the multithreading API.
3. Helping to determine and provide prior suggestion, the guideline document of the usage and limitation of the multithreading API to the customer if deadlock defects are non-strategic for project planning and not fixed.
4. Emphasizing the importance of understanding overall system behavior of the application to the multithreading API and resolve programming issue, including deadlock.
5. Helping the development manager to make a decision how the potential deadlock should be fixed and to plan resources that affect to financial cost of the project.

1.5. Publications

Several parts of our research have been selected to be presented in international conferences and published in the corresponding proceedings detailed in Appendix A.

1.6. Research Methodology

To achieve the research, there are 8 following steps.

1. Studying to understand about conditions of deadlock and deadlock occurrences in the Java programming field
2. Studying the deadlock detection algorithm
3. Choosing an appropriate algorithm to use in this research
4. Designing a deadlock detection algorithm and tool
5. Developing the deadlock detection algorithm and tool
6. Testing the deadlock detection algorithm and tool
7. Exercising the example of test cases using the deadlock detection tool and evaluating output of the research.
8. Summarizing and analyzing the result and providing suggestion. And finally publishing the thesis

1.7. Organization of the Thesis

The remainder of the thesis is organized into four chapters as follows:

Chapter II presents theoretical background including API, Multithreading API, Deadlock, Deadlock Example, Deadlock in the Multithreading API, Six Conditions of Deadlock, Three Types of Deadlock Analysis, Deadlock Detection Algorithms, Path Coverage and Deadlock Code Patterns and Literature Reviews.

Chapter III describes the approach of this thesis for the deadlock detection algorithm and tool.

Chapter IV presents the result from the algorithm and tool detecting the potential deadlock in the multithreading API and validation of the algorithm and tool.

Finally, Chapter V concludes research work and presents some directions for future work, Limitation of our work are also detailed.

CHAPTER II
BACKGROUND THEORY AND LITERATURE REVIEWS

2.1. Background Theory

2.1.1. Application Programming Interface (API)

In Java Tutorials website published by Oracle [7] stated that the API was a large collection of ready-made software components that provided many useful capabilities. It was grouped into libraries of related classes and interfaces; these libraries were known as packages.

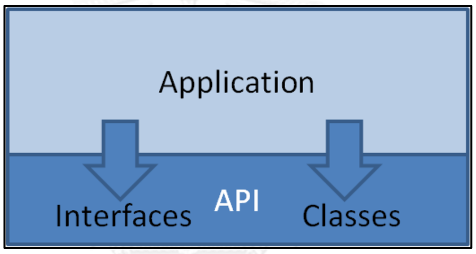


Figure 2 Application and API

Figure 2 illustrates the application calls Interfaces and Classes that are in the underlying API. For the application development, developers use Interfaces and Classes to develop the application. At runtime, the application will call Interfaces and Classes to work as it was developed.

2.1.2. Multithreading API

The multithreading API is developed for specific use on top of the general programming language such as the Java API. The multithreading API is the API that creates several threads to process their tasks when the application calls it at runtime.

Figure 3 illustrates the application is developed on top of the multithreading API and the multithreading API is developed on top of the Java API that is a general programming language.

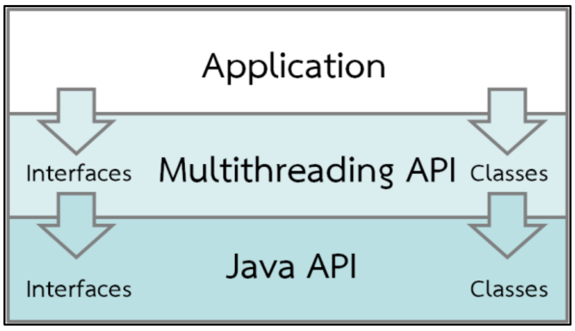
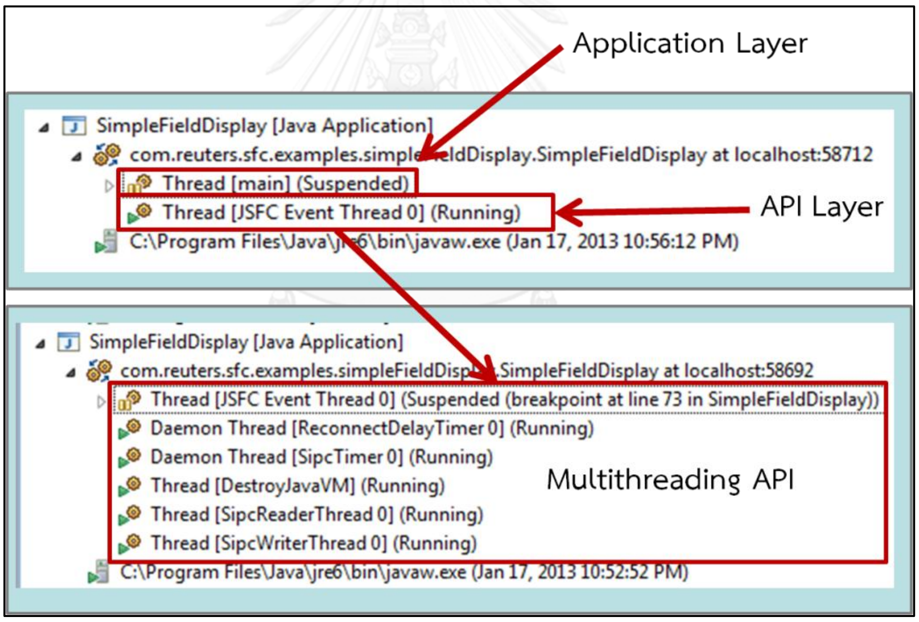


Figure 3 The application is implemented on top of the multithreading API

Figure 4 illustrates the stack trace of the application at runtime when the main thread of the application calls objects in the multithreading API, the multithreading API creates several threads to handle tasks as regards calling of the application.



```
SimpleFieldDisplay [Java Application]
  com.reuters.sfc.examples.simpleFieldDisplay.SimpleFieldDisplay at localhost:58712
    Thread [main] (Suspended)
    Thread [JSFC Event Thread 0] (Running)
  C:\Program Files\Java\jre6\bin\javaw.exe (Jan 17, 2013 10:56:12 PM)
```

```
SimpleFieldDisplay [Java Application]
  com.reuters.sfc.examples.simpleFieldDisplay.SimpleFieldDisplay at localhost:58692
    Thread [JSFC Event Thread 0] (Suspended (breakpoint at line 73 in SimpleFieldDisplay))
    Daemon Thread [ReconnectDelayTimer 0] (Running)
    Daemon Thread [SipcTimer 0] (Running)
    Thread [DestroyJavaVM] (Running)
    Thread [SipcReaderThread 0] (Running)
    Thread [SipcWriterThread 0] (Running)
  C:\Program Files\Java\jre6\bin\javaw.exe (Jan 17, 2013 10:52:52 PM)
```

Figure 4 The multithreading API

2.1.3. Deadlock

Oracle [8] defined that deadlock described a situation where two or more threads were blocked forever, waiting for each other.

Deadlock occurring are described as four deadlock conditions by Edward G. Coffman, Jr. in 1971 [9]. The deadlock occurs if following four conditions occurs simultaneously.

2.1.3.1. Mutual Exclusion

The involved resources must be unshareable. Only one process can use the resources anytime.

2.1.3.2. Hold and Wait

The processes must hold the resources they have already been allocated while waiting for other requested resources.

2.1.3.3. No Preemption

The process must not have resources taken away while those resources are being used.

2.1.3.4. Circular Wait

The process must be waiting for a resource which is being held by another process, which in turn is waiting for the first resource to release resource. In general, there is a set of waiting processes, P = { P1, P2, P3, ..., PN}, such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3 and so on until PN is waiting for a resource held by P1.

The following examples are deadlock examples in the Computer fields.

2.1.4. Deadlock Example

2.1.4.1. Alphones and Gaston Story [8]

There is a classic example that is always used as an example for the Java API proposed by Oracle. It is about Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time.

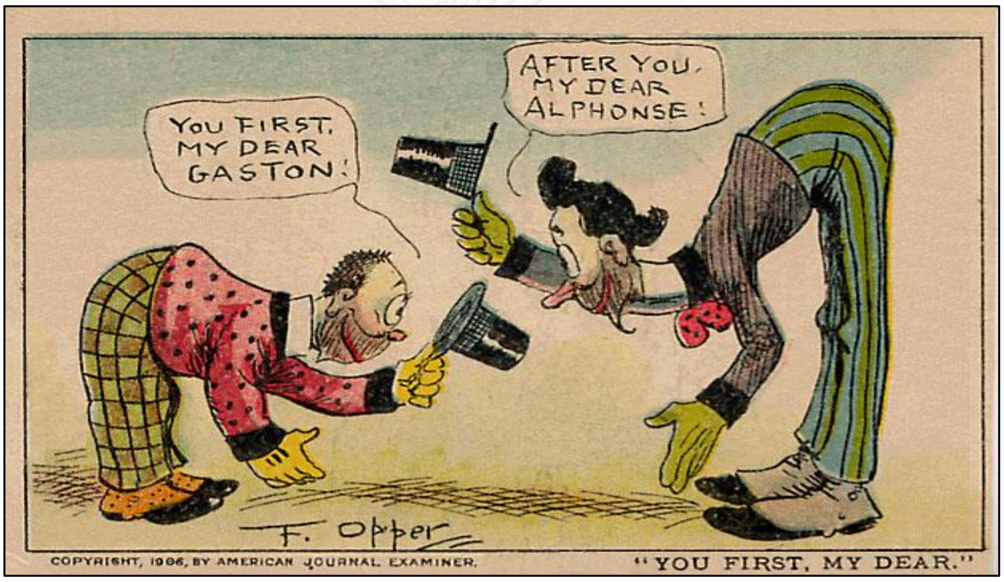

YOU FIRST, MY DEAR GASTON!
AFTER YOU, MY DEAR ALPHONSE!
F. Opper
COPYRIGHT, 1906, BY AMERICAN JOURNAL EXAMINER.
"YOU FIRST, MY DEAR."

Figure 5 Alphonse and Gaston Story for Deadlock Example [8]

This example application is shown in Table 1, Deadlock, models this possibility:

Table 1 The deadlock code example

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }
    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

When Deadlock runs, it's extremely likely that both threads will block when they attempt to invoke bowBack. Neither block will ever end, because each thread is waiting for the other to exit bow.

2.1.4.2. Some Deadlock Properties of Computer Systems

This is an example about the deadlock in the Computer System written by Richard C. HOLT in 1972 [4]. The deadlock occurs when there is more than one process in the Computer System. One process blocks another process and both cannot work as their desire. If there are two processes that are represented with P1 and P2 and two resources that are represented with R1 and R2. Assume that resource cannot be released if a process is waiting for a request. If R1 is used by P1 and R2 is used by P2. And P1 requests to use R2 concurrently with P2 requests to use R1. The result is P1 and P2 is deadlock.

2.1.4.3. Deadlock Detection in Distributed Databases

This example is the deadlock that occurs in the Database proposed by Edgar Knapp in 1987 [10]. The cause of deadlock in the database begins with there is sharing the same data in the Database. Several transactions work concurrently and request to access the same data in the Database. Assume that there is a data reading operation in the Database stored at x; R(x). And there is a data writing operation in the Database stored at x; W(x). When there are more than two transactions operate in the Database concurrently, the database transaction operates interleaved fashion. The interleaved fashion is the transaction that operates to use a resource concurrent with another transaction that operates to use another resource and then cross operating on resources. The interleaved fashion causes incorrect data to store in the Database after the transactions end. To solve the interleaved fashion of two processes work together on same data in the Database is ‘Locking’. Locking prevents other transactions to access data that is locked. However, if transactions cross accessing to data that are locked by each other, the deadlock occurs.

For these examples, as describe hereinbefore, exchanging resources between more than one process, there are resource blocking and requesting interleaving, it causes deadlock occur. The root cause of deadlock is incorrect synchronization of processes [4].

2.1.5. Deadlock in the multithreading API

In the big picture of the application, the API is also a part of the application. However, if the application is developed by 3rd parties that are not the same team that develop the API, the API should be separated from the application. Figure 6 illustrates the deadlock occurs in an API layer. The main thread is called by the application layer and locks object R1 and object R2 and it requires object R3. However, object R3 is locked by Thread A that is in the API Layer and Thread A also requires object R1 to process its task. This situation is common way for the deadlock in the API. Both threads also require the same resources; however each other thread already locks the resource and doesn't release the resource until they have done their tasks. Therefore both threads block and wait for each other caused 'Deadlock'.

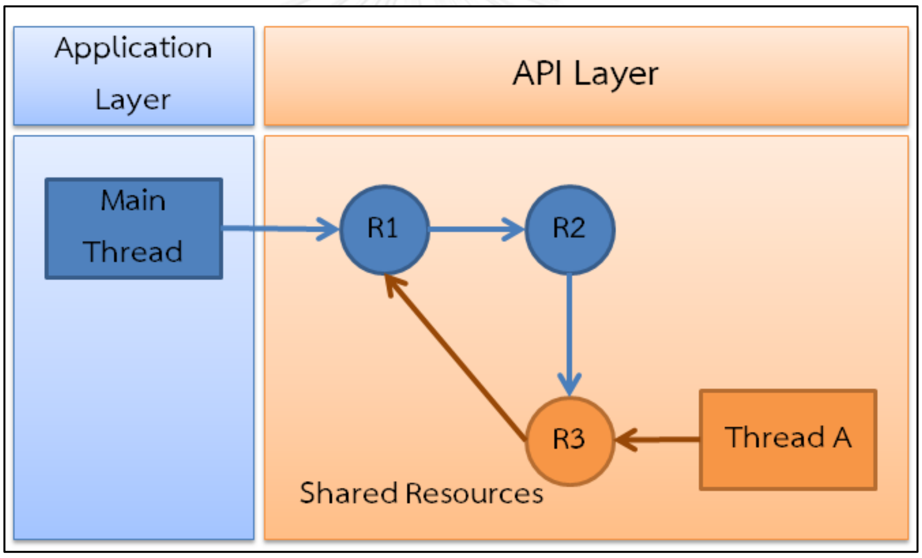
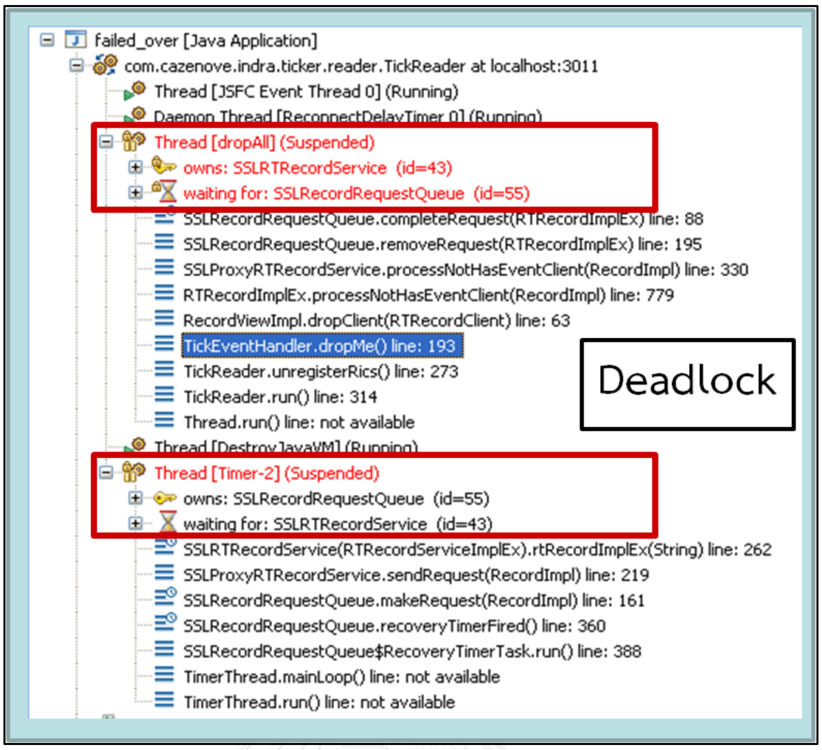


Figure 6 Deadlock in an API

The deadlock in the multithreading API is the API creates several threads when the application calls the API at runtime and some threads locks and waits to acquire the same objects. Figure 7 facilitates understanding for the deadlock in the multithreading API.



```
failed_over [Java Application]
  com.cazenove.indra.ticker.reader.TickReader at localhost:3011
    Thread [JSFC Event Thread 0] (Running)
    Daemon Thread [ReconnectDelayTimer 0] (Running)
    Thread [dropAll] (Suspended)
      owns: SSLRTRecordService  (id=43)
      waiting for: SSLRecordRequestQueue  (id=55)
      SSLRecordRequestQueue.completeRequest(RTRecordImplEx) line: 88
      SSLRecordRequestQueue.removeRequest(RTRecordImplEx) line: 195
      SSLProxyRTRecordService.processNotHasEventClient(RecordImpl) line: 330
      RTRecordImplEx.processNotHasEventClient(RecordImpl) line: 779
      RecordViewImpl.dropClient(RTRecordClient) line: 63
      TickEventHandler.dropMe() line: 193
      TickReader.unregisterRics() line: 273
      TickReader.run() line: 314
      Thread.run() line: not available
    Thread [DestroyJavaVM] (Running)
    Thread [Timer-2] (Suspended)
      owns: SSLRecordRequestQueue  (id=55)
      waiting for: SSLRTRecordService  (id=43)
      SSLRTRecordService(RTRecordServiceImplEx).rtRecordImplEx(String) line: 262
      SSLProxyRTRecordService.sendRequest(RecordImpl) line: 219
      SSLRecordRequestQueue.makeRequest(RecordImpl) line: 161
      SSLRecordRequestQueue.recoveryTimerFired() line: 360
      SSLRecordRequestQueue$RecoveryTimerTask.run() line: 388
      TimerThread.mainLoop() line: not available
      TimerThread.run() line: not available
```

Figure 7 Deadlock in the multithreading API

In the stack trace of deadlock in the multithreading API, Figure 7 shows that there are 2 threads lock and wait for the same objects. Above thread owns the SSLRTRecordService object and is waiting for the SSLRecordRequestQueue object when calling the completeRequest() method. Below thread owns the SSLRecordRequestQueue object and is waiting for the SSLRTRecordService object when calling the rtRecordImplEx() method. Therefore both threads encounters deadlock.

2.1.6. Six conditions of Deadlock

Referring to the Effective Static Deadlock Detection in 2009 proposed by Mayur Naik, David Gay, Chang-Seo Park and Koushik Sen [3], they have presented the following six conditions of deadlock that can be used to develop the deadlock detection algorithm.

2.1.6.1. Reachable Condition

In some execution of the program, can a thread abstracted by Thread A reach line 1 of A and, after acquiring a lock at line 1 of A, proceed to reach line 2 of A while still holding the lock (and similarly for Thread B reach line 1 of B and then line 2 of B)?

2.1.6.2. Aliasing Condition

In some execution of the program, can a lock acquired at line 1 of A be the same as a lock acquired at line 2 of B (and similarly for line 2 of A and the line 1 of B)?

2.1.6.3. Escaping Condition

In some execution of the program, can a lock acquired at line 1 of A be accessible from more than one thread (and similarly for each of line 2 of A and then line 1 of B and line 2 of B)?

2.1.6.4. Parallel Condition

In some execution of the program, can different threads abstracted by Thread A and Thread B simultaneously reach line 2 of A and line 2 of B, respectively?

2.1.6.5. Non-Reentrant Condition

In some execution of the program, can a thread abstracted by Thread A acquire a lock at line 1 of A it does not already hold and, while holding that lock, proceed to acquire a lock at line 2 of A it does not already hold (and similarly for thread B and then line 1 of B and then line 2 of B)? If the thread acquires the same lock it already holds then the second lock acquisition cannot cause a deadlock as locks are reentrant in Java.

2.1.6.6. Non-Guarded Condition

In some execution of the program, can different threads abstracted by Thread A and Thread B reach line 1 of A and line 1 of B, respectively, without holding a common lock? If the two threads already hold a common lock then we call it a guarding lock (also called a gate lock).

2.1.7. Three Types of Deadlock Analysis

There are three types that are used for deadlock analysis as follows:

2.1.7.1. Static Analysis

In 2005, Tong Li explained, the static analysis performed analysis on all possible control flow paths. However, considering all possible paths also forced static tools to face the issue of filtering out potentially large amounts of false positives [2]. Mattia Moga et al. added suggestion for the static analysis approach in 2009, the static analysis that performed flow-sensitive, interprocedural and context-sensitve data flow analysis was quite efficient and precise with low false positives rate [11]. This research aims to use this approach because the deadlock detection should be detected for all paths in the scope of the API prior to API released to the customer and we still don't know exactly scenarios and application usage of the customers.

2.1.7.2. Dynamic Analysis

Mayur has gathered the definition of dynamic analysis in 2009. He explained that the dynamic analysis was the deadlock actually occurred in execution. In addition, the dynamic approached monitor the program workflow in an execution and report cycles in the resulting dynamic graph [3]. Mattia Moga et al. believed that the dynamic analysis introduced a significant runtime overhead in the application being analyzed [11]. Tong Li et al. also added about the drawback of dynamic analysis, it was hard to reproduce the deadlock using dynamic analysis. We had to wait and run the code repeatedly to detect the deadlock. In addition, it only considered control flow paths actually taken that might be under expectation [2].

2.1.7.3. Hybrid Analysis

Mattia Monga et al. in 2009 proposed the hybrid analysis blended the strengths of static and dynamic analysis approaches. Static analysis considered the source code without actually executing it. The strength was that it could reason over all possible program paths but they were often overly conservative that normally reported properties weaker than dynamic analysis that actually held in a specific execution. Dynamic analysis focused on an actual execution of the target application that considered only a limited number of program paths (i.e., those that have been covered in the observed executions), but they could provide more accurate results. However, dynamic analysis introduced a significant runtime overhead in the application being analyzed. The hybrid analysis relied on a static preprocessing technique to reduce the runtime overhead of the subsequent dynamic analyzing. The static analysis identified dangerous statements and the dynamic analysis monitoring identified statement at runtime [11]. This approach still isn’t appropriate for deadlock detection in the API because it uses the dynamic analysis.

2.1.8. Deadlock Detection Algorithms

These are deadlock detection algorithms that use to detect the deadlock.

2.1.8.1. Directed Graph

Ahmed K. Elmagarmid described in year 1986 about the directed graph. In the directed graph, transactions and resources were represented by vertices and the requests and allocations by edges [12].

2.1.8.2. Wait-For Graph (WFG)

In 2005 Tong Li, C. S. E., Alvin R. Lebeck, and Daniel J. Sorin explained in their paper, the wait-for graph was a general resource graph. The nodes represented processes and edges represented dependences between processes. The usage for this graph was if there was an edge from node A to node B, it meant process A was waiting for process B to release a resource. A cycle in a WFG indicated a deadlock. Figure 8 shows a WFG for process A that is waiting for resource x from process B, process B is waiting for resource y from process C and process C is waiting for resource z from process A. The cycle in graph shows the deadlock. Constructing a WFG requires dynamically tracking the status of resources. It tracks the owner of each resource and the processes that are being waited for resources at a time [2].

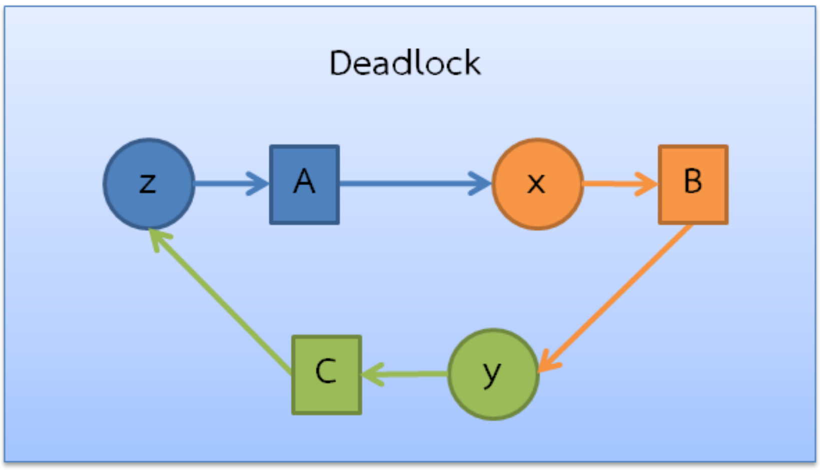


Figure 8 Wait-For Graph shows deadlock

2.1.8.3. Call Graph

Usman Ismail described about the call graph that it was defined as a set of directed edges connecting call site (statements invoking method calls) to corresponding target methods. The call graph was an analysis that can be used to help the programmers to understand and debug large programs. It supported all path coverage for the application. There were several call graph generating techniques such as Reachability Analysis (RA), Class Hierarchy Analysis (CHA) and Rapid Type Analysis (RTA). He suggested that the RTA was an algorithm that was able to create precise call graphs. RTA determined the set of class instantiated in the context of the call site and used this information to filter the number of possible target methods [13].

2.1.9. Path Coverage

Path coverage is the one of the most important criteria that is used to investigate the sufficiency of software testing. It requires that every path in a program should be executed at least once [14].

2.1.10. Deadlock Code Patterns

Frank Otto and Thomas Moschny presented code patterns indicating possible synchronization problems that encountered the deadlock [15].

2.1.10.1. Cyclic Lock Dependencies

The following two code fragments are shown in Table 2, which are both path of parallel application, acquire lock in different orders:

Table 2 The Cyclic Lock Dependency example

`void foo() { synchronized(o1) { synchronized(p1) { // … } } }`	`void bar() { synchronized(p2) { synchronized(o2) { // … } } }`

Problem: A cyclic lock dependency is the classic pre-condition of a deadlock. The existence of cyclic lock dependencies can be determined by considering the constraints of the program. The above example consists of two constraints o1 $\rightarrow$ p1, p2 $\rightarrow$ o2, such that there are cyclic lock dependencies provided that o1 in foo() and o2 in bar() as well as p1 in foo() and p2 in bar() may point to the same objects, respectively. In this case, a deadlock may occur when foo() and bar() are executed in parallel.

Detection:

1) Consider the set C of all constraints of the program.
2) Build a directed graph G = (V, E). We define vertexes V := C and edge E := {(c1 $\rightarrow$ c2, c'1 $\rightarrow$ c'2) $\in$ C X C : PS(c2) $\cap$ PS(c'1) $\neq$ Ø}. That is, each constraint is represented by a vertex, and an edge between two constraints says that the second lock of the first constraint may point to the same object as the first of the second constraint.
3) Find all cycles in G, e.g. by using a depth-first algorithm.
4) For each cycle, determine the set B of involved blocks.
5) If mhp(b1, b2) = true for all b1, b2 $\in$ B, report the cycle.

2.1.10.2. Superfluous Lock

Consider the body of a synchronized block, although it never access shared objects concurrently:

Table 3 The Superfluous Lock example

```
synchronized(o) {
      // only non-critical operations
      // …
}
```

<u>Problem:</u> A superfluous lock is a lock protecting some code that only performs "non-critical" operations. We define an operation to be non-critical if it never accesses any shared object concurrently. A superfluous lock may result in efficient synchronization by making other threads unnecessarily block.

<u>Detection:</u> For each synchronized block b of the program:

1) Consider the set B of block that may be executed in parallel to b, i.e. B := {b' : mph(b, b') = true}.
2) For each block b' ∈ B, check if b and b' might access shared objects such that there could be read-write- or write-write-conflicts.
3) If this is not the case, then b is probably unnecessarily protected. Report b.

2.2. Literature Reviews

The following are related works have researched about deadlock detection.

2.2.1. Effective Static Deadlock Detection

Firstly in this research, they defined six types of deadlock occurrence that was in the preceding part of this proposal. They used Discrete Mathematics to represent methods, variables and application flows. And propose the deadlock detection algorithm using a tool called JADE (Java Agent DEvelopment framework).

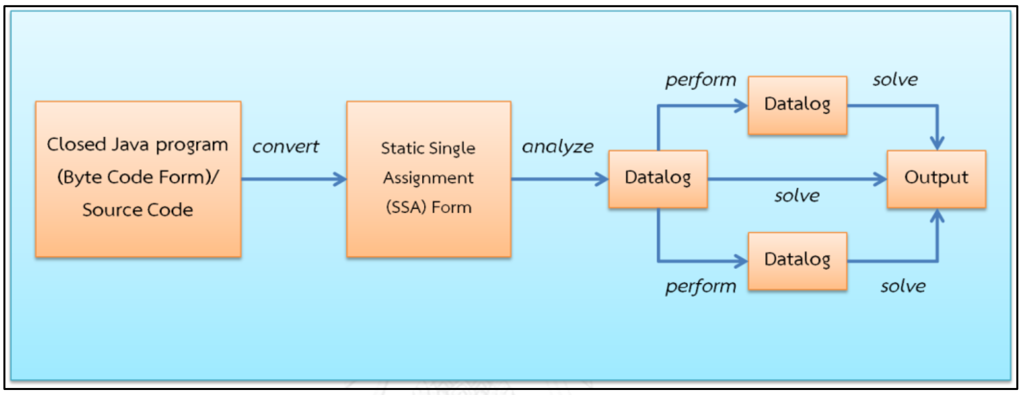

Closed Java program (Byte Code Form)/ Source Code
convert
Static Single Assignment (SSA) Form
analyze
Datalog
perform
Datalog
solve
Output
solve
perform
Datalog
solve

Figure 9 Implementation of the Deadlock Detection Algorithm

The deadlock detection algorithm is implemented by a tool called JADE. The input of the algorithm can be the closed Java program in the byte code form or the source code. Then it uses the Soot framework to construct 0-CFA-based call graph (CFA stands for Control Flow Analysis) and get the result with set M of methods that may be reachable from the main method. Then rewrite each synchronized block to synchronize on argument v with body s. And then change format to SSA (Static Single Assignment) to increase the precision of the flow-insensitive k-object-sensitive analysis. The result from k-object-sensitive analysis; Datalog, is performed by the thread-escape and may-happen-in-parallel analysis. Results from three analyses called Datalogs will be solved using BDD (Binary Decision Diagram) based the Datalog solver. The outputs from BDDs are the program information, the analyzed output, the relation finalDeadlock and the set of paths categorized by six types of deadlock occurrence.

This paper suggests that the static analysis is effective for developing the deadlock detection tool for multithreading Java programs comprising over 1.5 MLOC in practice. They also added that using the dynamic analysis approaches the deadlock could occur in a different execution. It is unsound and cannot be applied to open programs and without the input test data for simulating the different scenario of execution [3].

Therefore this thesis will use the static analysis to develop the deadlock detection algorithm.

2.2.2. Finding Synchronization Defects in Java Programs: Extended Static Analyses and Code Patterns

This paper was presented by Frank Otto and Thomas Moschny in 2008. They proposed an approach to indicate the possible synchronization problems using static analysis combined with point-to and may-happen-in-parallel (MHP) analysis to detect the code patterns that may result in deadlocks or race condition or indicate inefficient synchronization in the Java programs. In addition, focusing on reduce false positive [15].

In our thesis we will use the deadlock patterns suggested by this paper to implement the deadlock detection algorithm to find deadlocks in the source code of the multithreading API.

2.2.3. Understanding Complex Multithreaded Software Systems by Using Trace Visualization

Jonas Trümper proposes the tool to help the developers to understand and envision the flow and behavior of software that is developed previously using visualization concept. The visualization concept that they proposed, they provide the graph to show the mechanism inside the application at runtime consists of threads. Their tool analyzes the software system and captures threads' activities and represents them as a graph. They call the behavior of tool that analyzes the software system behavior at runtime that dynamic analysis. Referring to the visualization concept, it can help the developers to understand the flow and behavior of software [16].

In the same way as the deadlock detection, it demonstrates the behavior of the multithreading API with the graph such as the wait-for graph or the call graph, is able to facilitate the deadlock analysis. In our thesis the deadlock detection tool should report the result of the deadlock sites for increasing understanding of developers for the multithreading API behavior and guiding them for future resolution of deadlock.

2.2.4. Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables

Rahul Agarwal and Scott D. Stoller in 2006 presented concepts to develop the programs to detect potential deadlocks at runtime. The program detects the deadlock that occurs in the event for the following operation: acquire and release of locks, wait- notify on condition variables, up and down operations on semaphores, accesses to shared variables and thread start, join and termination operations. They implemented the program using their multithread GoodLock algorithm detects potential for deadlock to handle block and using the Bensalem and the Havelund's algorithms to handle non-block structured locking. They used dynamic analysis to implement the program to detect the deadlock because they expected few false alarms that should be eliminated from infeasible paths for using static analysis [17].

In this thesis we detect deadlock that occurs in the following operation: acquire and release of locks using synchronization statement and wait-notify condition on variables.

2.2.5. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution

Tong Li, Carla S. Elis, Alvin R. Lebeck and Daniel J. Sorin proposed a novel operating system mechanism to detect deadlock dynamically in applications in 2005. It scans the system for processes that have been blocked using Speculative execution to discover dependencies amongst the sleeping processes and to determine that applications encounter deadlocks. It constructs a general resource graph and checks for the graph that contains cycles will be reported as deadlocks. It also detects various types of deadlock. They also added that Pulse can use with other tools to expand accuracy of deadlock coverage [2].

2.2.6. Static Deadlock Detection for Java Libraries

This paper proposed by A.Williams, W.Thies, and M.D. Ernst in 2005. They presented a flow-sensitive, context-sensitive analysis for static deadlock detection in Java libraries. They used lock-order graphs to represent locking in Java libraries. In the graph, nodes are sets, edged are lock orderings and cycles indicate deadlocks. They focused on deadlock occurred by synchronized statements and the wait-notify methods of Java. They also added that using static analysis is more accurate than using dynamic analysis because of all path coverage; however, there are some spurious reports because of infeasible flows [18].

This paper supports our approach for this thesis to use static analysis for the deadlock detection algorithm. In addition, they support for detecting the deadlock from the synchronization statements and wait-notify methods.

2.2.7. Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients

This paper was proposed by Jyotimoy Deshmukh, E.Allen Emerson and Sriram Sankaranarayanan in 2009. They presented the Symbolic Deadlock Analysis to detect deadlock based on synchronization for re-entrant locks. The tool separates deadlock identifying in Java libraries, concurrent libraries and the multithreaded client applications for decoupling the deadlock reports in each part. The tool was developed on the static analysis concept. It uses lock-order graph analysis, logical formulae for symbolic enumeration of alias patterns, soot framework, May-aliases for tracking lock objects across methods. The final result of this tool still has fault positives and redundant deadlock causing alias patterns [19].

Their symbolic concept is interesting; however, it provides fault positive and duplicate deadlock report.

2.2.8. Ant Colony Optimization for Deadlock Detection in Concurrent Systems

This paper proposed by Gianpiero Francesca, Antonella Santone, Gigliola Vaglini and Maria Luisa Villani in 2011. They used Calculus of Communicating System (CCS) to detect deadlock in the source code. CCS is one of a temporal-logic formula representing the requirement to be verified as regards the concept of Model checking. The Model checking is used to verify complex systems and is able to prove correctness of a system. The complex system has general problems such as deadlock. In addition, the problem occurs in the system with many components interact with each other or in system with data structures that can assume many different values. They proposed the use of Ant Colony Optimization (ACO) to reduce the state space explosion problem. ACO algorithms are stochastic techniques. They compare the result of several optimization algorithms that are the A* algorithm, ACO, Breadth-First Search (BFS) and Concurrency Workbench of New Century (CWB-NC). The final result is the ACO algorithm provides best result of minimal counterexamples [20].

Using the Model checking concept to develop the deadlock detection algorithm is interesting; however, the Model checking returns very long counterexamples. The ACO algorithm helps to reduce the state space explosion and provide short counterexamples but the ACO are still stochastic techniques. The ACO uses estimation from the structure of the process can cause imprecise result. Moreover, it is possible that the ACO loses some paths to get shortest counterexamples. The result of stopping running when solution found filters other solutions more than half. The ACO can optimize the result with faster searching but the accuracy is dropped and in fact in the Development phase of finding deadlock needs all path coverage.

CHAPTER III
APPROACH OF THIS THESIS

From the general concepts of four deadlock conditions by Edward G. Coffman, Jr. [9], there are two researches that enhance this concept to create their own static deadlock detection algorithms to detect deadlocks in Java source codes. The first research was presented a static deadlock detection algorithm that has six deadlock conditions by Mayur Naik in 2009 [3] and another research was presented the detection algorithm for finding synchronization defects that has two related conditions to detect deadlocks by Frank Otto in 2008 [15]. We select five conditions from the first research that are in the scope of our research. We do not include the Reentrance Condition into our Deadlock Detection Algorithm because our scope of our research does not consider detecting deadlock in the multithreading API that is developed by using java.util.concurrent package. And we select two conditions from the second research to develop our own algorithm to detect deadlocks in the multithreading API. The Deadlock Detection Algorithm detects deadlock in the multithreading API and provides information which conditions encounter deadlock to help developers understand the behavior of the multithreading API and deadlock occurance. Figure 10 shows derivation of our research.

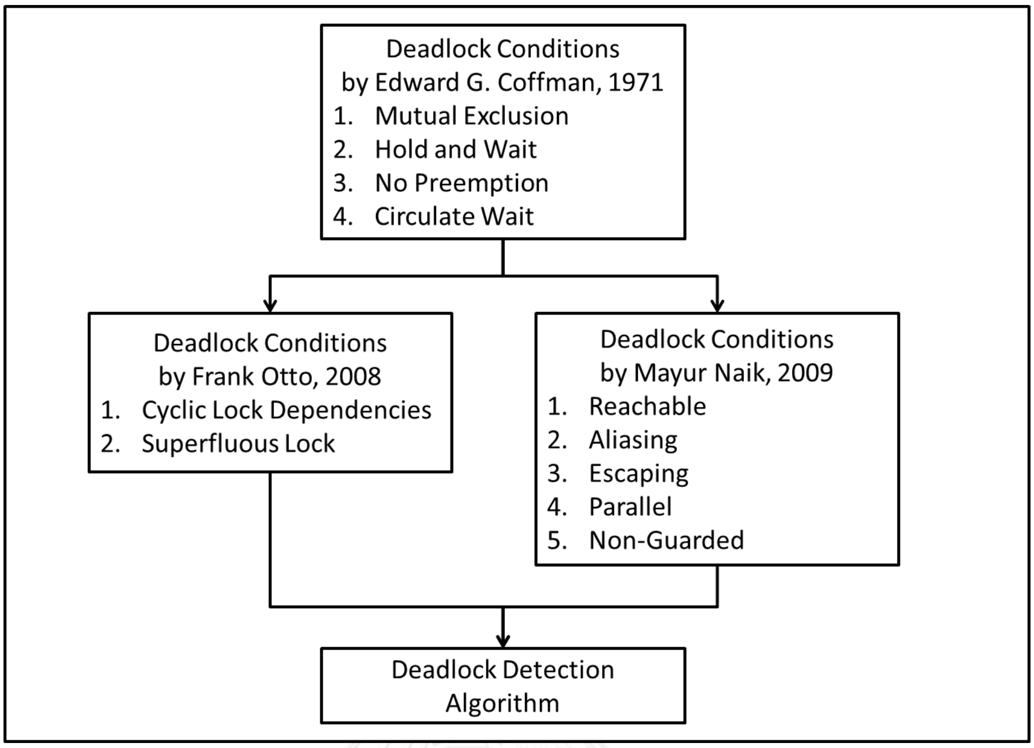


Figure 10 Derivation of the algorithm

In addition, we develop the prototype tool for deadlock detection that uses the Deadlock Detection Algorithm to demonstrate the development of the Deadlock Detection Algorithm. We use the Java Programming Language to develop the Deadlock Detection Algorithm and Tool. We design the Deadlock Detection Tool to get the Java source codes that have several files as an input of the Deadlock Detection Tool. We create Java files of source code to validate the result of the Deadlock Detection Algorithm. Each file is represented as a multithreading API and solely finishes its work in its file. Therefore all files do not relate or call to other files and a file is for testing a scenario of deadlock occurance. The Deadlock Detection Tool reports number of files that are found deadlock, absolute path of each file for informing location to developers, deadlock site for each file, number of Threads that are created in each file and deadlock conditions of each file. Figure 11 shows overall of the Deadlock Detection Tool.

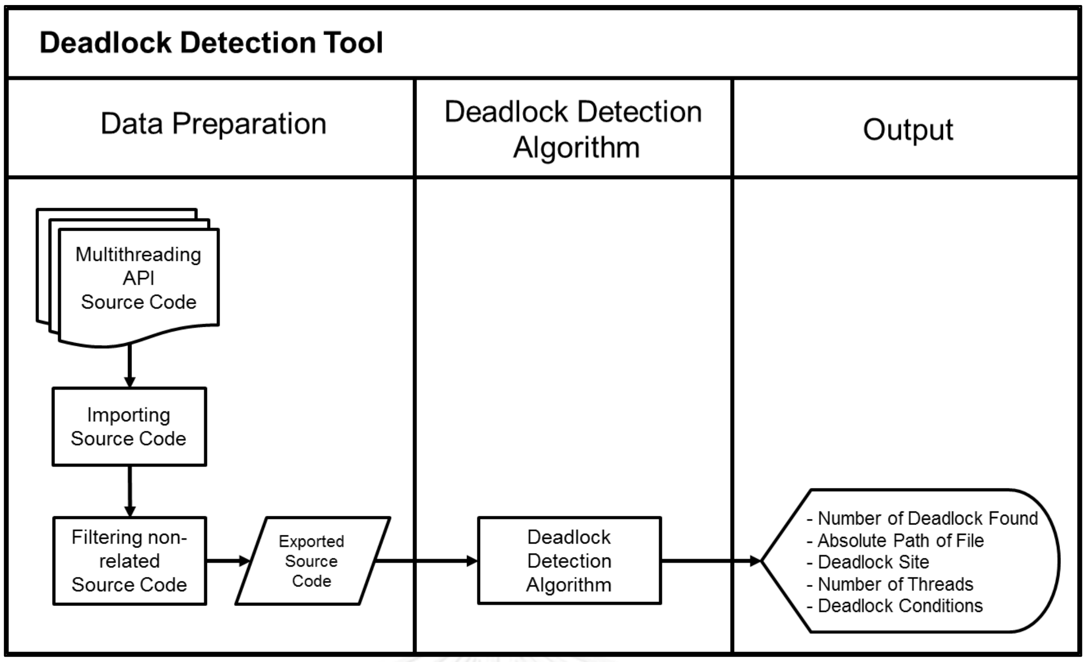


Figure 11 The Deadlock Detection Tool

We describe more detail about the Deadlock Detection Algorithm and Tool in the following sections.

3.1. Deadlock Detection Algorithm

Our Deadlock Detection Algorithm is derived from five in six deadlock conditions [3] and from two in five code patterns of deadlock conditions [15] to develop our own algorithm to detect deadlock in the multithreading API source code. Figure 12 shows the Deadlock Detection Algorithm diagram.

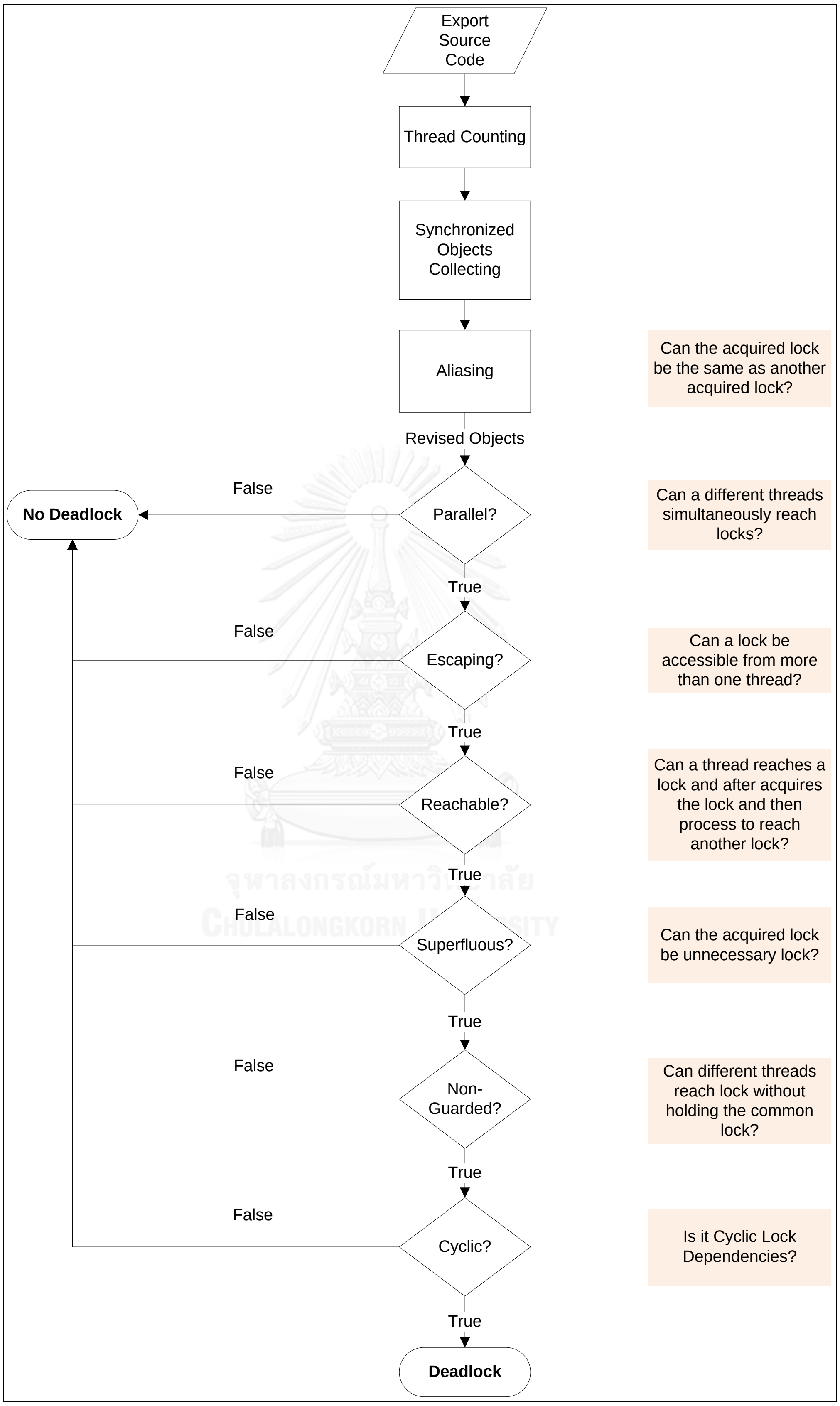

Export Source Code
Thread Counting
Synchronized Objects Collecting
Aliasing
Can the acquired lock be the same as another acquired lock?
Revised Objects
False
No Deadlock
Parallel?
Can a different threads simultaneously reach locks?
True
False
Escaping?
Can a lock be accessible from more than one thread?
True
False
Reachable?
Can a thread reaches a lock and after acquires the lock and then process to reach another lock?
True
False
Superfluous?
Can the acquired lock be unnecessary lock?
True
False
Non-Guarded?
Can different threads reach lock without holding the common lock?
True
False
Cyclic?
Is it Cyclic Lock Dependencies?
True
Deadlock

Figure 12 The Deadlock Detection Algorithm diagram

In Figure 12 the algorithm starts after the input of the source code that is exported to the algorithm by filtering out the non-related deadlock source code that are the source code comments and source codes that are not in the synchronization block, to focus on the source code that relates to deadlock. The focus source code are the synchronized keyword, the start() method called by Thread Class, the synchronization block, Object declaration, Object reference, the Thread Class extension and the Runnable interface implementation. The algorithm counts the number of threads that are created. After that it collects all synchronized objects from the exported source code. Next it processes the Aliasing Condition. If the acquired lock is the same as another acquired lock, the collected objects are changed to the reference one. If the acquired lock is not the same as another acquired lock, the collected objects remain the same. Then it checks the Parallel Condition. If there are different threads are created simultaneously and reach locks, this condition returns true and there is deadlock possibly occurs. If there is no different threads are created simultaneously and reach locks, this condition returns false and there is no deadlock occurs. And after that the algorithm checks the Escaping Condition. If there is a lock is accessed by more than one thread, this condition returns true and there is deadlock possibly occurs. However, if there is no lock is accessed by more than one thread, this condition returns false and there is no deadlock occurs. And then it checks the Reachable Condition. If a thread reaches a lock and after acquires the lock and then process to reach another lock, this condition returns true and there is deadlock possibly occurs deadlock. However, if it occurs in the other hand, this condition returns false and there is no deadlock occurs. And then it checks the Superfluous Lock Condition. If the acquired locks are redundant, this condition returns true and deadlock possibly occurs. In the other hand, this condition returns false and there is no deadlock occurs. After that it checks the Non-Guarded Lock Condition. If different threads reach locks without holding a common lock, this condition returns true and there is deadlock possible occurs. However, if different thread reaches a lock and already holds a common lock, this condition returns false and there is no deadlock occurs. Finally, algorithm checks the Reverse Order Locking Condition or the Cyclic Lock Dependency

Condition. If the order of locking is reverse or cyclic lock dependencies, it returns true and there is deadlock possibly occurs. In other hand, if the other of locking is not reverse or cyclic lock dependencies, it returns false and there is no deadlock occurs.

The following explain in detail for each block of the algorithm diagram.

3.1.1. Exported Code

The exported code is the input from the multithreading API source code that is filtered out non-related source code. The non-related source codes are comments and the part of codes that are not synchronization blocks or keywords. The related deadlock source code are the synchronized keyword, the start() method keyword of the Thread Class, the synchronization block, Object declaration, Object reference, the Thread Class extension and the Runnable interface implementation. This input is an output from the Data Preparation process. The exported source code is used as an input of the Deadlock Detection Algorithm. We will deep down in detail for the Data Preparation process in the Deadlock Detection Tool.

3.1.2. Thread Counting

Deadlock occurs when there is more than one Thread lock the same Objects and with reverse order of more than two Objects. Therefore we count the number of Thread created in the source code.

In the Deadlock Detection Algorithm, the exported codes; java files, are read one by one and the algorithm checks each line of files of source codes whether there are ".start()" String or not. If the line of code has the ".start() String, the algorithm counts up the thread number integer by one. If there are more than one Thread call the start() method, it is possible that deadlock occurs. If there is only one Thread calls the start() method, deadlock does not occur. Table 4 shows the pseudo code of the Thread Counting method in the Deadlock Detection Algorithm.

Table 4 The pseudo code of the Thread Counting of the Deadlock Detection Algorithm

```
For (File exportedFile : exportedSource.listFiles())
      If exportedFile.isDirectory
            Call method countThreads with exportedFile
      Else
            Initialise exportedFileName to
exportedFile.getAbsolutePath
            Initialise exportedClassName to exportedFile.getName
            If exportedClassName.contains with ".java"
                  Create new File
                  Initialise threadCount to 0
                  Initialise br to null
                  Try
                        Create new BufferedReader
                        Initialise line to null
                        Try
                              While ( line )
                                    If line.contains with
".start()"
                                          increment threadCount
                                    EndIf
                              EndWhile
                        Catch IOException e
                              Call method e.printStackTrace
                        EndTry
                  Catch FileNotFoundException e
                        Call method e.printStackTrace
                  Finally
                        If br is not equal to null
                              Try
                                    Call method br.close
                              Catch IOException e
                                    Call method e.printStackTrace
                              EndTry
                        EndIf
                  EndTry
            EndIf
      EndIf
EndFor
```

3.1.3. Synchronized Object Collecting

We collect synchronized Objects in the source code to check each condition whether there is deadlock in the source code or not.

In the algorithm the synchronized Objects are counted for each exported files. The algorithm compares each line of exported java files with “synchronized” String and then adds the synchronized Object name in the ArrayList includes braces to show its synchronized block. We use the ArrayList for collecting synchronized Objects because we can add synchronized Objects sequentially to the ArrayList when reading the source code and it is easy to access to the ArrayList. After the algorithm finishes collecting all synchronized Objects in each files of exported source code, the algorithm processes each condition that are the Aliasing Condition, the Parallel Condition, the Escaping Condition, the Reachable Condition, the Superfluous Lock Condition, the Non-Guarded Lock Condition and the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition sequentially. We will elaborate more detail of each condition in the later section. Table 5 shows the pseudo code of the Synchronized Object Collecting process in the Deadlock Detection Algorithm.

Table 5 The pseudo code of the Synchronized Object Collecting method in the deadlock detection algorithm

```
For (File exportedSourceFile : exportedSource.listFiles())
      If exportedSourceFile.isDirectory
            Call method collectSyncObjs with exportedSourceFile
      Else
            If exportedSourceFile.getName
                  Create new ArrayList
                  Initialise syncCountThis to 0
                  Initialise pw to null
                  Initialise br to null
                  Initialise exportedSourceFileName to
exportedSourceFile
                  Initialise syncInfoFileName to "D:\\DDSyncInfo\\"
                  Try
                        Create new PrintWriter
                        Create new BufferedReader
                        Initialise line to ""
                        While ( line )
                              If ( line.contains with
"synchronized(" )
                                    If line.contains with
"synchronized(this)"
                                          SyncCountThis++;
                                          Call method syncObjs.add
with "this"
                                    Else if line.contains with
"synchronized("
                                          Initialise syncObjName to
line.substring
                                          Call method syncObjs.add
with syncObjName
                                    Else if line.contains with
"synchronized"
                                          Initialise syncObjName to
line.substring
                                          Set syncObjName to
syncObjName.substring
                                          Call method syncObjs.add
with syncObjName
                                    EndIf
                              Else if line.contains with "{"
                                    Call method syncObjs.add with
"{"
                              Else if line.contains with "}"
                                    Call method syncObjs.add with
"}"
                              EndIf
                        EndWhile
                        Call method pw.println with syncObjs

                        Set parallelCondBool to parallelCond with
syncInfoFile.getAbsolutePath, syncObjs

                        Set escapingCondBool to escapingCond with
syncInfoFile.getAbsolutePath, syncObjs

                        Set reachableCondBool to reachableCond with
syncInfoFile.getAbsolutePath

                        Initialise revisedSyncObjs
                        to aliasingCond with
exportedSourceFile.getAbsolutePath, syncObjs

                        Set superfluousCondBool to superfluousCond
with exportedSourceFile.getAbsolutePath, revisedSyncObjs
                        Set nonguardedCondBool to nonGuardedCond
with exportedSourceFile.getAbsolutePath, revisedSyncObjs

                        Set cyclicCondBool to cyclicCond with
exportedSourceFile.getAbsolutePath, revisedSyncObjs

                  Catch IOException e

                  Finally
                        If pw is not equal to null
                              Call method pw.close
                        EndIf
                  EndTry
            EndIf
      EndIf
EndFor
```

3.1.4. Aliasing Condition

The Aliasing Condition is the condition that a lock and another lock are referring to the same object. For example; if Thread A acquires Object L1 and then acquires Object L2 sequentially. Object L1 is a reference of Object L2. Object L1 and Object L2 are the aliasing objects. Figure 13 shows the example of the Aliasing Condition.

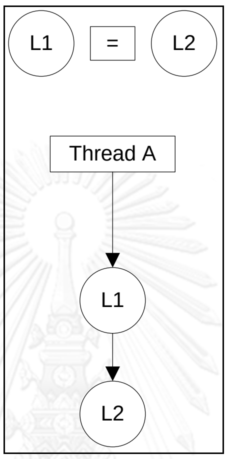


Figure 13 The example of the Aliasing Condition

Figure 14 shows another case for the Aliasing Condition. In another case, Thread A acquires Object L3. Thread B acquires Object L4. Object L3 and Object L4 are aliasing objects.

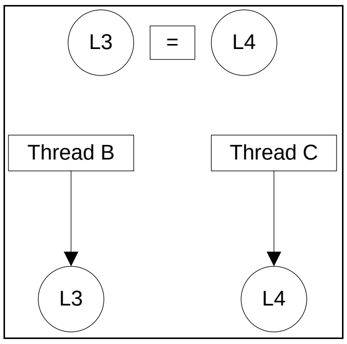


Figure 14 The example of the Aliasing Condition

For our algorithm, in the Aliasing Condition, the algorithm checks whether the locked Objects are aliasing or not. If the locked objects are aliasing, the condition returns true. If the locked objects are not aliasing, the condition returns false. Both cases of the Aliasing Condition possibly encounter deadlock. For example; if Thread A acquires Object L1 and Object L2 sequentially and Thread D acquired Object L2 and Object L1 sequentially. This example deadlock should occur but if Object L1 and Object L2 are aliasing therefore the deadlock does not occur. Figure 15 shows the Aliasing Condition that deadlock does not occur.

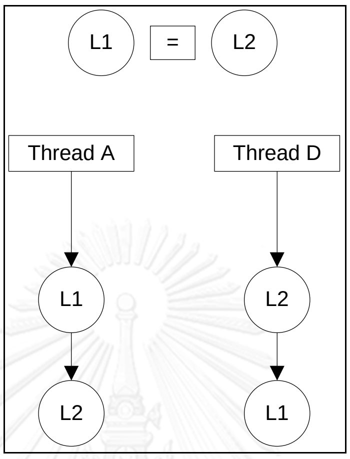


Figure 15 The Aliasing Condition that deadlock does not occur

In another case, if Thread B acquires Object L3 and Object L1 sequentially and Thread C acquires Object L1 and Object L4 sequentially. This example deadlock should not occur because it does not have the cyclic object locking but Object L3 and Object L4 are aliasing therefore the deadlock occurs. Figure 16 shows the example of the Aliasing Condition that deadlock occurs.

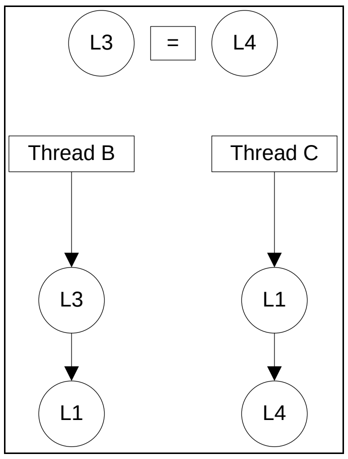


Figure 16 The Aliasing Condition that deadlock occurs

For the source code of the Aliasing Condition in the Deadlock Detection Algorithm we collect reference Objects in the HashMap and refer object as key that is not duplicate and its alias as a value. And after that the algorithm checks the keys of Object in the HashMap whether they are in the ArrayList of the synchronized Objects or not. The algorithm considers only Objects in the HashMap that are in the ArrayList and then transfers keys and values to another HashMap. After that the algorithm creates a new ArrayList to copy the synchronized objects and replaces them with their aliasing Objects. The algorithm sets the flags for the Aliasing Condition if the ArrayList of synchronized Object is replaced with its aliasing and returns the ArrayList that has latest revised synchronized Objects to process to the next condition that is the Parallel Condition. Table 6 shows pseudo code of the Aliasing condition of the Deadlock Detection Algorithm.

Table 6 The psuedo code of the Aliasing Condition of the Deadlock Detection Algorithm

```
Create new HashMap
      Initialise br to null
      Initialise line to ""
      Initialise lineSplitString to null
      Initialise newLength to 0
      Try
            Create new BufferedReader
            Try
                  While ( line )
                        If line.contains with "=" and not line.contains with "new"
                              Set lineSplitString to line.split with " "
                              For i is 0, i is less than lineSplitString.length, i increments by 1
                                    If position i in lineSplitString
                                          Set newLength to i

                                          Call method System.arraycopy with lineSplitString
                                          If position 2 in aliasingLineSplitString
                                                Set 2 of aliasingLineSplitString to position 2 in aliasingLineSplitString
                                          EndIf


                                          If position 2 in aliasingLineSplitString
```

```
                                                        Set 2 of
aliasingLineSplitString to position 2 in aliasingLineSplitString
                                                 EndIf
                                                 Call method
aliasingObjs.put
                                          EndIf
                                   EndFor

                            Else if line.contains with "=" and
line.contains with "new"
                                   Set lineSplitString to line.split
with " "
                                   Create new String array of length 2
                                   For i is 0, i is less than
lineSplitString.length, i increments by 1
                                          If position i in
lineSplitString
                                                 Set 0 of tempArray to
position in lineSplitString
                                                 Set 1 of tempArray to
position in lineSplitString
                                          EndIf
                                   EndFor
                                   If position 1 in tempArray
                                          Set 1 of tempArray to position
1 in tempArray
                                   EndIf

                                   If position 1 in tempArray
                                          Set 1 of tempArray to position
1 in tempArray
                                   EndIf

                                   Call method aliasingObjs.put with
position 0 in tempArray, position 1 in tempArray
                            EndIf
                     EndWhile
              Catch IOException e

              EndTry
       Catch FileNotFoundException e

       Finally
              If br is not equal to null
                     Try
                            Call method br.close
                     Catch IOException e

                     EndTry
              EndIf
       EndTry
EndIf

Create new HashMap
For (String s : syncObjs)
       For (Map.Entry<String, String> entry :
aliasingObjs.entrySet())
              If entry.getKey
```

```
                  Call method aliasingObjsTemp.put with
entry.getKey, entry.getValue
            EndIf
      EndFor
EndFor

Create new HashMap
For (Map.Entry<String, String> entry : aliasingObjsTemp.entrySet())
      If ( not entry.getKey )
            Call method aliasingObjsTemp2.put with entry.getKey,
entry.getValue
      EndIf
EndFor

Initialise aliasingFlag to false
If not aliasingObjsTemp2.isEmpty

      Set aliasingFlag to true
Else

      Set aliasingFlag to false
EndIf

Create new ArrayList
If aliasingFlag
      For (Map.Entry<String, String> entry :
aliasingObjsTemp2.entrySet())
            For i is 0, i is less than syncObjs.size, i increments
by 1
                  If entry.getKey
                        Call method syncObjsTemp.add with i,
entry.getValue
                  Else
                        Call method syncObjsTemp.add with i,
syncObjs.get i
                  EndIf
            EndFor
      EndFor
      If syncObjsTemp.size is greater than syncObjs.size
            Create new ArrayList
      EndIf

Else
      Set syncObjsTemp to syncObjs

EndIf
Set aliasingCondBool to aliasingFlag

If aliasingFlag
            Call method pwAliasingCondTrue.append with file plus "
\n"
EndIf
Return syncObjsTemp
```

3.1.5. Parallel Condition

The Parallel Condition is the condition that there are several Threads are created and run simultaneously and each Thread locks Objects. For example; in the program there are 2 Threads are created and run simultaneously that are Thread A and Thread B. Thread A locks Object L1 and locks Object L2 sequentially. Thread B locks Object L3. Figure 17 shows the example of the Parallel Condition.

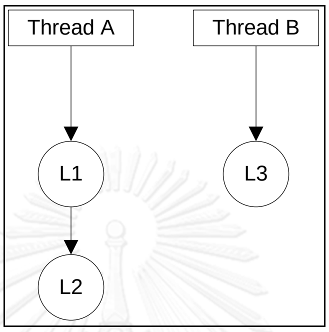


Figure 17 The example of the Parallel Condition

For our algorithm, after getting the ArrayList of the synchronized Object names of each files, the algorithm checks whether there are several threads created and each thread locks synchronized Objects or not. If in the files there are several threads created and lock synchronized Objects simultaneously, the algorithm returns true, the deadlock probably occurs and the algorithm goes to check for a next condition that is the Escaping Condition. However, if there are several threads created but do not lock synchronized Objects simultaneously or there is only one thread created and locks synchronized Objects, the algorithm returns false and does not check other conditions. Table 7 shows the pseudo code of the Parallel Condition of the Deadlock Detection Algorithm.

Table 7 The pseudo code of the Parallel Condition of the Deadlock Detection Algorithm

```
Initialise openBlock to 0
Initialise closeBlock to 0
Initialise lockedObj to 0
Initialise br to null
Initialise line to ""
Initialise numOfThread to 0
Initialise fileSearch to file.substring with file.lastIndexOf
"\\", +1
Initialise scanner to null
Try
      Create new BufferedReader
      Create new Scanner
      While scanner.hasNext
            If scanner.next
                  Call method scanner.next
                  If scanner.hasNextInt
                        Set numOfThread to scanner.nextInt
                  EndIf
            EndIf
      EndWhile
      For (String entry : syncObj)
            If entry.equals with "{"
            Else if entry.equals with "}"
            Else
                  lockedObj++;
            EndIf
      EndFor
      If ( numOfThread is greater than 1 ) and ( numOfThread is
not equal to 0 ) and (lockedObj is greater than or equal to
numOfThread )
            Call method pwPCondTrue.append with fileSearch plus
" \n"
            Set parallelCondBool to true
      Else
            Set parallelCondBool to false
      EndIf
Catch FileNotFoundException e
Finally
      If br is not equal to null
            Call method br.close
      EndIf
      Call method scanner.close
EndTry
Return parallelCondBool
```

3.1.6. Escaping Condition

The Escaping Condition is the condition that a lock can be accessible from more than one thread. For example; there is Object L1, it can be accessible by Thread A and Thread B. Figure 18 shows the example of the Escaping Condition.

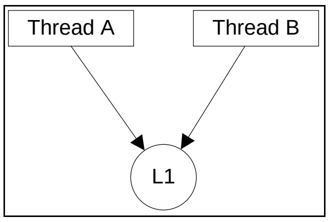


Figure 18 The example of the Escaping Condition

For our algorithm the Escaping Condition checks whether locks in java files can be accessible from more than one thread or not. If a lock can be accessible from more than one thread, the algorithm returns the Escaping Condition as a true value and the algorithm continues to execute the Reachable Condition. If a lock is not accessible from more than one thread, the algorithm returns the Escaping Condition as a false value. The algorithm counts synchronized Objects in the ArrayList and it adds synchronized Objects to the HashMap Object to check duplicate Objects. (We use the HashMap Object because the HashMap Object can check uniqueness of the Objects.) If synchronized Objects are duplicated, it means Obects are accessed by more than one thread and it is possible that deadlock occurs. The Escaping Condition returns true and continues to execute the next condition that is the Reachable Condition. If not, the Escaping Condition returns false and not check the next condition. Table 8 shows the pseudo code of the Escaping Condition of the Deadlock Detection Algorithm.

Table 8 The pseudo code of the Escaping Condition of the Deadlock Detection Algorithm

```
Create new HashMap
For (String entry : syncObj)
      Initialise count to map.get with entry
      Call method map.put with entry, ( count is equal to null )
EndFor


For (Map.Entry<String, Integer> entry : map.entrySet())
      If not ( entry.getKey )
            If entry.getValue is greater than 0
                  Call method pwEsCondTrue.append with file plus
" \n"
                  Set escapingCondBool to true
            EndIf
      EndIf
EndFor
```

3.1.7. Reachable Condition

The Reachable Condition is the condition that a Thread can reach to a lock and acquire it and then reach to another lock and is still holding the first lock. For example; Thread A reaches to Object L1 and acquires Object L1 and then it reaches to Object L2. Thread A is still holding Object L1 when reaches to Object L2. Figure 19 shows the example of the Reachable Condition.

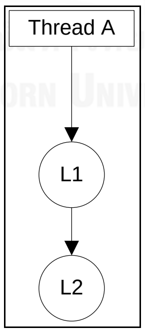


Figure 19 The example of the Reachable Condition

For our algorithm the Reachable Condition, the algorithm checks whether threads reach a lock and after acquiring the lock, they reach to another lock or not. We count the synchronized objects and get the number of threads. If the number of threads is more than 0 and the duplication of synchronized objects is more than one, it means that there are several threads reach several objects. The algorithm returns true and continues to execute the next condition that is the Aliasing Condition. However, if the condition returns false, the algorithm does not execute the next condition.

Table 9 shows the pseudo code of the Reachable Condition of the Deadlock Detection Algorithm.

Table 9 The pseudo code of the Reachable Condition of the Deadlock Detection Algorithm

```
Initialise numOfThread to 0
Create new BufferedReader
Create new Scanner
While scanner.hasNext
      If scanner.next
            Call method scanner.next
            If scanner.hasNextInt
                  Set numOfThread to scanner.nextInt
            EndIf
      EndIf
EndWhile
Create new HashMap
For (String entry : syncObj)
      Initialise count to map.get with entry
      Call method map.put with entry, ( count is equal to null )
EndFor
For (Map.Entry<String, Integer> entry : map.entrySet())
      If not ( entry.getKey )
            If numOfThread is greater than 0 and entry.getValue is
greater than 1
                  Call method pwReachableCondTrue.append with file
plus " \n"
            EndIf
      EndIf
EndFor
If br is not equal to null
            Call method br.close
EndIf
```

3.1.8. Superfluous Lock Condition

The Superfluous Lock Condition is the condition that the synchronized Objects are unnecessary duplicate locked. For example; Thread A acquires Object L1 and after that in the source code Thread A calls to acquires Object L1 again. Thread A acquires Object L1 twice that is unnecessary duplicate lock. The Superfluous Lock can cause the system unnecessary block an Object and lead to deadlock. Figure 20 shows the example of Superfluous Lock Condition.

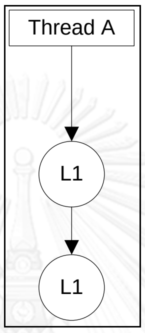


Figure 20 The example of the Superfluous Lock Condition

In our algorithm, the Superfluous Lock Condition, it checks locked Objects of each Thread whether each Thread locks duplicate Object or not. If yes, the deadlock does not occur. If no, the deadlock potentially occurs. The algorithm checks adjacent Objects in the ArrayList of synchronized Objects. If both adjacent Objects are equal, it is a Superfluous Lock. If both adjacent Objects are not equal, it is not a Superfluous Lock. The result of the adjacent Objects is collected in another ArrayList. If all elements of the result ArrayList of the adjacent Objects are equal, the deadlock potentially occurs. If not all elements of the result ArrayList of the adjacent Objects are not equal, the deadlock does not occur. Table 10 shows the pseudo code of the Superfluous Lock Condition.

Table 10 The pseudo code of the Superfluous Lock Condition of the Deadlock Detection Algorithm

```
SuperfluousLockCond
Initialise superfluous to false

      For i is 0, i is less than revisedSyncObjs.size, i increments
by 1
            If ( i is equal to ( revisedSyncObjs.size decrements by
1)
            break;
            Else
                  If ( revisedSyncObjs.get i equals with "{" or
revisedSyncObjs.get i equals with "}" or revisedSyncObjs.get i
increments by 1 equals with "}" or revisedSyncObjs.get i increments
by 1 equals with "{" )
                  continue;
                  Else
                  index.add with i
                  doubleLockList.add ( Call method
arrayListToken.get (
revisedSyncObjs.get i, revisedSyncObjs.get i increments by 1)
                  EndIf
            EndIf
      EndFor
      Initialise countTrue to 0

      For(boolean boolEle : doubleLockList)
            If booEle equal to true
                  increment countTrue
            EndIf
      EndFor

      If countTrue is greater than 0
            Set superfluousBool to true
      Else
            Set superfluousBool to false
            Call method pwSuperflousCondTrue.append with
absolutePath plus "\n")
      EndIf

Return superfluousBool




arrayListToken (a, b)
Initialise doubleLock to true
If a equal b
      Set doubleLock to true
ElseIf
      Set doubleLock to false
EndIf
```

3.1.9. Non-Guarded Lock Condition

The Non-Guarded Lock Condition is the invert condition of the Guarded Lock Condition. The Guarded Condition helps to solve deadlock problems by adding the same synchronized Object before the couple of reverse order of synchronized Objects that are locked by threads. The synchronized Object is called the Guarded Lock or the common lock. The Guarded Lock prevents deadlock occurrence. When the Thread acquires the Guarded Lock, other thread cannot hold that Guarded Lock until the Thread releases the Guarded Lock. Therefore the reverse order of synchronized Objects of other threads cannot access until the first thread that acquires the Guarded Lock finishes the task and releases the Guarded Lock. For example; if there are Thread A and Thread D. Thread A has acquired sequence of Objects that are L4, L1 and L2 and Thread B has acquired sequence of Objects that are L4, L2 and L1. This scenario is showed in Figure 21 in the left call graph it does not occur deadlock because when Thread A acquires L4, Thread D has to wait Thread A until Thread A releases L4. And then Thread A owns L4 and acquires L1 and L2 and Thread D still waits for L4. When Thread A releases L2 and Thread A still owns L4 and L1, Thread D still waits for L4. When Thread A releases L1, it owns L4 and Thread D still waits for L4. After that when Thread A releases L4, Thread D can acquire L4. The deadlock does not occur for the left call graph scenario. For the right call graph scenario, there is no Guarded Lock before the couple of reverse order of synchronized objects; L1 and L2 therefore the deadlock occurs.

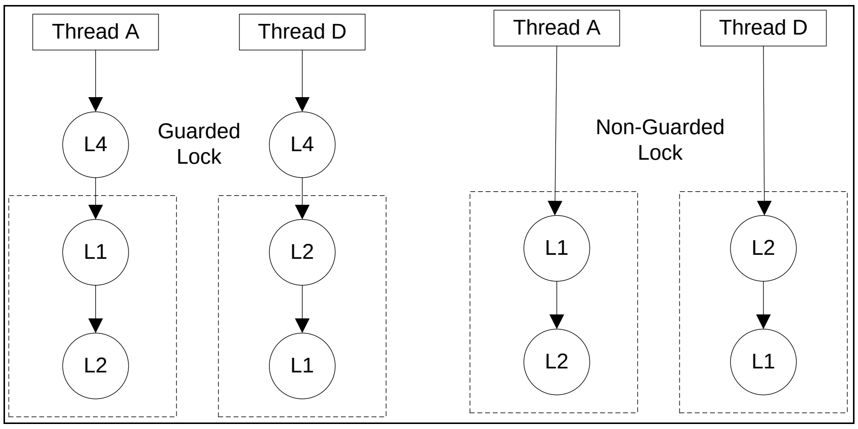


Figure 21 The left call graphs show Guarded Lock Condition and the right call graphs show Non-Guarded Lock Condition

For our algorithm the Non-Guarded Lock Condition checks whether the first lock Object locked by each Thread is the same Object or not. If the first lock Object is the same as each Thread, it is a Guarded Lock and the Non-Guarded Lock Condition returns false. In the other hand, if the first lock Object is not the same for each Thread, there is no Guarded Lock and the Non-Guarded Lock Condition returns true. Table 11 shows the pseudo code of the Non-Guarded Lock Condition.

Table 11 The pseudo code of the Non-Guarded Lock Condition of the Deadlock Detection Algorithm

```
Set nonguarded to false
      Initialise groupMaxSize to 0
      For i is 0, i less than index.size, i increments by 1
            Set groupMaxSize to Call method
checkGroupSize(index.get with i, index.get with i increments by 1
increments by groupMaxSize
      EndFor
      If numOfThread is equal to 2
            If groupMaxSize is equal to 2
                  If revisedSyncObjs.get with index.get with 0
equals( revisedSyncObjs.get with index.get with groupMaxSize
                        Set nonguarded to false
                  Else
                        Set nonguarded to true
                  EndIf
            Else if groupMaxSize is equal to 3
                  If index.size is equal to 5
                        If ( index.get with groupMaxSize decrements
by 1 ) increments by 1 is equal to index.get with groupMaxSize
                              Set groupMaxSize to groupMaxSize
decrements by 1
                        If ( revisedSyncObjs.get with (index.get
with 0) equals to revisedSyncObjs.get with index.get with
groupMaxSize      )                                Set nonguarded to
false                                              Else if
                              Set nonguarded to true
                              Else
                              If ( revisedSyncObjs.get with
index.get with 0) equals ( revisedSyncObjs.get with index.get with
groupMaxSize )
                                    Set nonguarded to false
                              Else
                                    Set nonguarded to true
                              EndIf
                        EndIf

                  Else if index.size equal to 6
                        If ( revisedSyncObjs.get with index.get(0))
equals ( revisedSyncObjs.get with index.get with groupMaxSize ))
                              Set nonguarded to false
                        Else
                              Set nonguarded to true
                        EndIf
                  EndIf
            EndIf

      Else if numOfThread is equal 3
            Set groupMaxSize to groupMaxSize decrements by 1
                  If ( revisedSyncObjs.get with index.get with 0)
equals( revisedSyncObjs.get with index.get with groupMaxSize ))
                        If ( revisedSyncObjs.get with index.get
with 0 equals( revisedSyncObjs.get with index.get with groupMaxSize
increments by 2 ))
                              Set nonguarded to false
                        Else
                              Set nonguarded to true
                        EndIf
                  Else
                        Set nonguarded to true
                  EndIf
      Else
            Set nonguarded to true
      EndIf
If nonguarded
      pwNonGuardedCondTrue.append with absolutePath plus " \n"
EndIf
Return nonguarded
```

We scope our Deadlock Detection Algorithm to have capability to detect deadlock in the multithreading API source code that uses Oracle9i Application Server Best Practices [21]. The best practices avoid or minimize using of synchronization in the source code. The code should not have threads more than 3 and nested synchronization block should not more than 2 to avoid overhead cost and deadlock. Therefore the Non-Guarded Lock Condition has the condition to detect 2 and 3 threads and the number of locks of each thread is 2 and 3. The algorithm checks the first index of both Threads whether they are the same or not. If they are the same, it is the Guarded Lock and deadlock does not occur. If not, it is the Non-Guarded Lock and deadlock occurs and the algorithm continues to the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition.

If the number of Threads is 2 and the number of locks is 2 and 3, the algorithm checks the first index of both Threads whether they are the same or not. If yes, it is the Guarded Lock and deadlock does not occur. If not, it is the Non-Guarded Lock and deadlock occurs and the algorithm continues to the the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition.

If the number of Threads is 3 and the number of locks is 2, the algorithm checks the first index of 3 Threads whether they are the same or not. If they are the same, it is the Guarded Lock and deadlock does not occur. If not, it is the Non-Guarded Lock and the deadlock occurs and the algorithm continues to check the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition. And for other conditions, deadlock potentially occurs and the algorithm continues to check the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition.

3.1.10. Reverse Order Locking Condition or Cyclic Lock Dependency Condition

The Reverse Order Locking Condition or the Cyclic Lock Dependency Condition is a condition that the synchronized Objects locked by two or more than two threads are reverse. For example, in Figure 22 shows Thread A acquires L1 and L2 and Thread D acquires L2 and L1 sequentially. The lock order of Thread A and Thread D is reverse. It causes deadlock. When Thread A owns L1 and Thread D owns L2, after that Thread A waits to acquire L2 but L2 is owned by Thread D that waits to acquire L1 that is owned by Thread A. Both threads lock Objects in reverse order.

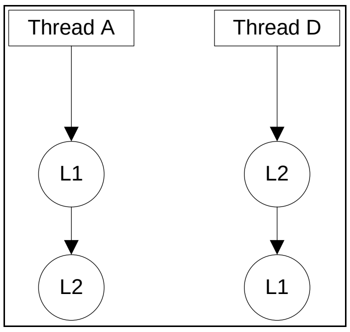


Figure 22 The Reverse Order Locking Condition or the Cyclic Lock Dependency Condition that deadlock occurs

Figure 23 shows the Wait-For Graph that has a Cyclic Lock Dependency Condition. It occurs deadlock because Thread A waits to acquire L2 but it owns L1 and Thread D waits to acquire L1 but it owns L2.

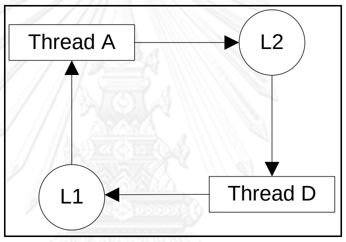


Figure 23 The Wait-For Graph shows deadlock of a Cyclic Lock Dependency Condition

In our algorithm, the Cyclic Lock Dependency Condition also implements to have capability to detect deadlock in the multithreading API source code that uses Oracle9i Application Server Best Practices [21] as well.

For this condition, first of all the algorithm ignores the synchronized 'this' because in Java when locking 'this' it does not occur deadlock. Then it gets the number of Threads that are created. And then counts the number of locks of each Thread. If Thread number is 2 and each Thread locks an Object, the deadlock cannot occur. If the first Thread locks 3 Objects and the second Thread locks 2 Objects, the deadlock does not occur when the first lock of each Thread are the same. The deadlock occurs when the first lock of first Thread and the last lock of the second Thread are the same and the second or the third lock of the first Thread are the same lock as the first lock of the second Thread.

For the first Thread locks 2 Objects and the second Thread locks 3 Objects, if the first lock of the first Thread and the first lock of the second Thread are the same, the deadlock does not occur. However, if the first lock of the first Thread and the second or the third lock of the second Thread are the same and the second lock of the first Thread and the first lock or the second lock of the second Thread are the same, the deadlock occurs.

For the case that there are 2 threads and each Thread lock 2 Objects, if the first lock of the first Thread are the same as the second lock of the second Thread and the second lock of the first Thread are the same as the first lock of the second Thread, the deadlock occurs. If there are 2 threads and each Thread lock 3 Objects, if the first lock of both Threads are the same, deadlock does not occur. If the first lock of the first Thread are the same as the second or the third lock of the second Thread and the second or the third lock of the first Thread are the same as the third or the second lock of the second Thread sequentially, the deadlock occurs.

For the case that there are 3 threads and each Thread lock 2 Objects, if the first lock of 3 Threads are the same, the deadlock does not occur. If the first lock of the first Thread are the same as the second lock of the third Thread, the second lock of the first lock and the first lock of the second Thread are the same and the second lock of the second Thread and the first lock of the third Thread are the same, the deadlock occurs. If the first lock of the first Thread and the second lock of the second Thread are the same, the second lock of the first Thread and the first lock of the third Thread are the same and the first lock of the second Thread and the second lock of the third Thread are the same, the deadlock occurs. If there are reverse order of 2 threads, the deadlock can also occur. For other case beside these, deadlock does not occur. The pseudo code of the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition is shown in Table 12.

Table 12 The pseudo code of the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition of the Deadlock Detection Algorithm

```
Set cyclicCondResult to false
Initialise index
      If parallelCondBool2 and escapingCondBool2 and
reachableCondBool2
            and not superfluousCondBool2 and nonguardedCondBool2
                  For i is 0, i less than revisedSyncObjs.size, i
increments by 1
                        If revisedSyncObjs.get with i equals to
"this"
                              revisedSyncObjs.set with i, "}"
                        EndIf
                  EndFor
                  For i is 0, i less than revisedSyncObjs.size, i
increments by 1
                        If not(revisedSyncObjs.get with i equals to
"{" or revisedSyncObjs.get with i  equals to "}"
                              index.add with i
                        EndIf
                  EndFor

                  Initialise br to null
                  Initialise line to ""
                  Initilaise numOfThread to 0
                  Initilaise fileSearch to absolutePath.substring
with
                              absolutePath.lastIndexOf with ("\\")
increments by 1, absolutePath.length

                        Initilaise scanner to null
                        Create new BufferedReader

                        While ( scanner.hasNext )
                              If (scanner.next equals to fileSearch
)
                                    scanner.next
                                    If (scanner.hasNextInt )
                                          Set numOfThread to
scanner.nextInt
                                          break;
                                    EndIf
                              EndIf
                        EndWhile

                  Set groupMaxSize to 0

                  For i is 0, i less than index.size, i increments
by 1

                        Set groupMaxSize to Call method
checkGroupSize with index.get with i, index.get with i increments
by 1

                        If numOfThread is equal to 2
                              If groupMaxSize is equal to 1
```

```
                                    Set cyclicCondResult to false

                              Else if groupMaxSize is equal to 3
and index.size is equal to 5
                                    If index.get with groupMaxSize
decrements by 1 increments by 1 less than index.get with
groupMaxSize
                                          If revisedSyncObjs.get
with index.get with 0 equals to revisedSyncObjs.get with index.get
with groupMaxSize
                                                Set
cyclicCondResult to false
                                          Else
                                                If
revisedSyncObjs.get with index.get with 0 equals to
revisedSyncObjs.get with index.get with index.size decrements by 1

                                                            If
revisedSyncObjs.get with

      index.get with index.size decrements by 2 equals to
revisedSyncObjs.get with index.get with 1 or revisedSyncObjs.get
with index.get with index.size decrements with 2 equals to
revisedSyncObjs.get with index.get with 2

                                                            Set
cyclicCondResult to true

                                                      Else
                                                            Set
cyclicCondResult to false

                                                      EndIf
                                                Else if
revisedSyncObjs.get with index.get with 1 equals to
revisedSyncObjs.get with index.get with index.size decrements by 1
and revisedSyncObjs.get with index.get with groupMaxSize decrements
by 1 equals to revisedSyncObjs.get with index.get with groupMaxSize

                                                      Set
cyclicCondResult to true
                                                Else
                                                      Set
cyclicCondResult to false
                                          EndIf
                                    EndIf
                              Else
                                    Set firstGroupSize to
groupMaxSize decrements by 1
                                    If revisedSyncObjs.get with
index.get with 0 equals to revisedSyncObjs.get with index.get with
firstGroupSize
                                          Set cyclicCondResult to
false

                                    Else if not revisedSyncObjs.get
with index.get with 0 equals to
revisedSyncObjs.get with index.get with firstGroupSize
```

```
                                          If revisedSyncObjs.get
with index.get with 0 equals to revisedSyncObjs.get with index.get
with index.size decrements by 2 or revisedSyncObjs.get with
index.get with 0 equals to revisedSyncObjs.get with index.get with
index.size decrements by 1 and revisedSyncObjs.get with index.get
with firstGroupSize decrements by 1 equals to revisedSyncObjs.get
with index.get with firstGroupSize
                                                Set
cyclicCondResult to true

                                          Else if
revisedSyncObjs.get with index.get with 0 equals to
revisedSyncObjs.get with index.get with index.size decrements by 1
and revisedSyncObjs.get with index.get with firstGroupSize
decrements by 1 equals to revisedSyncObjs.get with index.get with
firstGroupSize or revisedSyncObjs.get with index.get with
firstGroupSize decrements by 1 equals to revisedSyncObjs.get with
index.get with index.size decrements by 2
                                                Set
cyclicCondResult to true
                                          Else

                                                Set
cyclicCondResult to false
                                          EndIf
                                    EndIf

                              EndIf
                        Else if groupMaxSize is equal to 2 and
index.size is equal to 4
                              If revisedSyncObjs.get with index.get
with 0 equals to revisedSyncObjs.get with index.get with
groupMaxSize

                                    Set cyclicCondResult to false

                              Else
                                    If revisedSyncObjs.get with
index.get with 0 equals to revisedSyncObjs.get with index.get with
index.size decrements by 1 and revisedSyncObjs.get with index.get
with 1 equals to revisedSyncObjs.get with index.get with
groupMaxSize

                                          Set cyclicCondResult to
true
                                    Else
                                          Set cyclicCondResult to
false

                                    EndIf
                              EndIf
                        Else if groupMaxSize is equal to 3 and
index.size is equal to 6
                              If revisedSyncObjs.get with index.get
with 0 equals to revisedSyncObjs.get with index.get with
```

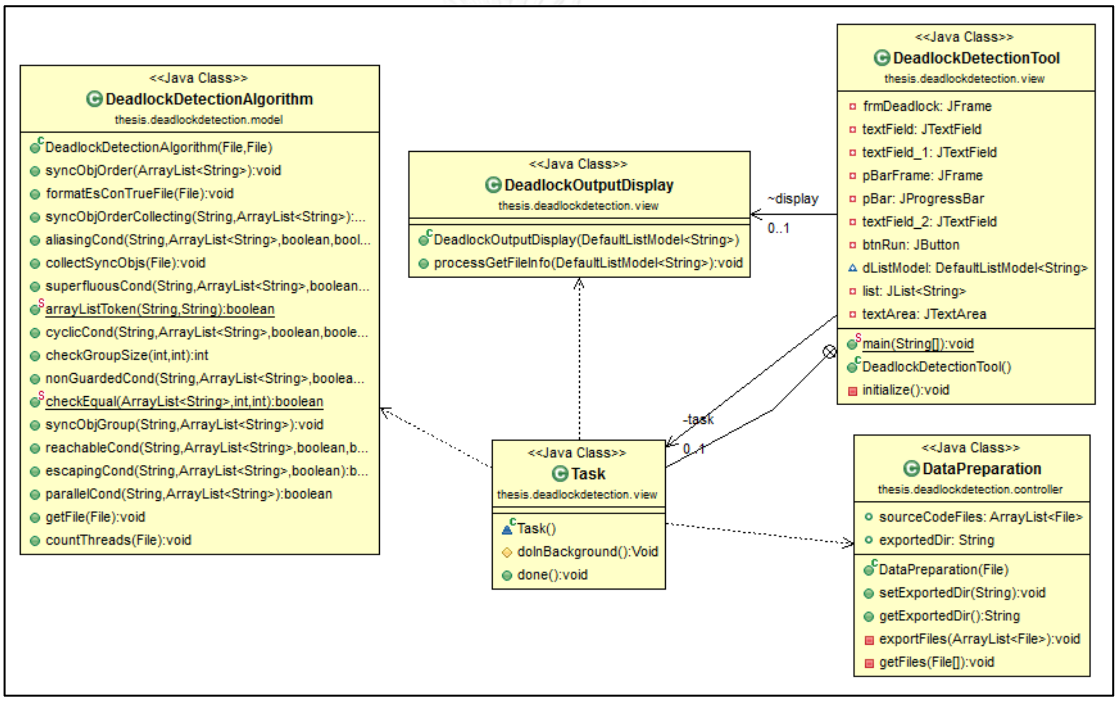
```
groupMaxSize

                                    Set cyclicCondResult to false
                              Else if not revisedSyncObjs.get with
index.get with 0 equals to revisedSyncObjs.get with index.get with
groupMaxSize

                                    If revisedSyncObjs.get with
index.get with 0 equals to revisedSyncObjs.get with index.get with
index.size decrements to 1 and revisedSyncObjs.get with index.get
with 1 equals to revisedSyncObjs.get with index.get with
groupMaxSize or revisedSyncObjs.get with index.get with 1
equals to revisedSyncObjs.get with index.get with groupMaxSize
increments by 1 and revisedSyncObjs.get with index.get with 2
equals to revisedSyncObjs.get with index.get with groupMaxSize or
revisedSyncObjs .get with index.get with 2 equals to
revisedSyncObjs.get with index.get with groupMaxSize increments by
1
                                          Set cyclicCondResult to
true
                                    Else if revisedSyncObjs.get
with index.get with 0 equals to revisedSyncObjs.get with index.get
with groupMaxSize increments 1 and revisedSyncObjs.get with
index.get with 1 equals
revisedSyncObjs.get with index.get with groupMaxSize or
revisedSyncObjs.get with index.get with 1
.equals to revisedSyncObjs.get with index.get with index.size
decrements by 1 and revisedSyncObjs.get with index.get with 2
equals to revisedSyncObjs.get with index.get with groupMaxSize or
revisedSyncObjs .get with index.get with 2 equals to
revisedSyncObjs.get with index.get with index.size decrements by 1
                                          Set cyclicCondResult to
true
                                    Else
                                          Set cyclicCondResult to
false
                                    EndIf
                              EndIf
                        EndIf
                  Else if numOfThread is equal to 3
                        If groupMaxSize is equal to 3 and
index.size is equal to 6
                              If index.get with groupMaxSize less
than  index.get with groupMaxSize increments by 1

                                    Set currentGroupSize to
groupMaxSize decrements by 1
                                    If revisedSyncObjs.get with
index.get with 0 equals to revisedSyncObjs.get with index.get with
2 equals to revisedSyncObjs.get with index.get with 4
                                          Set cyclicCondResult to
false
                                    Else
                                          If revisedSyncObjs.get
with index.get with 0 equals to revisedSyncObjs.get with index.get
with index.size decrements by 1 and revisedSyncObjs.get with
index.get with 1 equals to revisedSyncObjs.get with index.get with
2 and revisedSyncObjs.get with index.get with 3 equals to
```

```
revisedSyncObjs.get with index.get with 4
                                                Set
cyclicCondResult to true
                                          Else if
revisedSyncObjs.get with index.get with 0 equals to
revisedSyncObjs.get with index.get with 3 and revisedSyncObjs.get
with index.get with 1 equals to revisedSyncObjs.get with index.get
with 4 and revisedSyncObjs.get with index.get with 2 equals to
revisedSyncObjs.get with index.get with 5
                                                Set
cyclicCondResult to true
                                          Else if
revisedSyncObjs.get with index.get with 2
equals to revisedSyncObjs.get with index.get with 5 and
revisedSyncObjs.get with index.get with 3 equals to
revisedSyncObjs.get with index.get with 4

      Set cyclicCondResult to true
                                          Else if
revisedSyncObjs.get with index.get with 0
equals to revisedSyncObjs.get with index.get with 5 and
revisedSyncObjs.get with index.get with 1 equals to
revisedSyncObjs.get with index.get with 4
                                                Set
cyclicCondResult to true
                                          Else if
revisedSyncObjs.get with index.get with 0
equals to revisedSyncObjs.get with index.get with 3 and
revisedSyncObjs.get with index.get with 1 equals to
revisedSyncObjs.get with index.get with 2
                                                Set
cyclicCondResult to true
                                          Else
                                                Set
cyclicCondResult to false
                                          EndIf
                                    EndIf
                              EndIf
                        Else if index.size is equal to 5
                              If revisedSyncObjs.get with index.get
with 0 equals to revisedSyncObjs.get with index.get with index.size
decrements by 1
                                    If revisedSyncObjs.get with
index.get with 1 equals to revisedSyncObjs.get with index.get with
index.size decrements by 2
                                          Set cyclicCondResult to
true
                                    Else
                                          Set cyclicCondResult to
false
                                    EndIf
                              Else
                                    Set cyclicCondResult to false
                              EndIf
                        EndIf
                  Else if numOfThread greater than 3
                        If revisedSyncObjs.get with index.get with
0 equals to revisedSyncObjs.get with index.get with index.size
```

```
decrements 1
                              If revisedSyncObjs.get with index.get
with 1 equals to revisedSyncObjs.get with index.get with
groupMaxSize

                                    Set cyclicCondResult to true
                              Else
                                    Set cyclicCondResult to false
                              EndIf
                        Else
                              Set cyclicCondResult to false
                        EndIf
                  EndIf
            Else
                  Set cyclicCondResult to false
            EndIf
            If cyclicCondResult
                  Call to method pwCyclicCondTrue.append with
absolutePath plus " \n"
            EndIf
            Return cyclicCondResult
```

3.2. Deadlock Detection Tool

We develop the Deadlock Detection Tool using the Model-View-Controller (MVC) design pattern. The MVC design patten facilitates and makes tool development to be easier. For the Model, it is a part of the Deadlock Detection Algorithm. For the View, it is a Graphic User Interface (GUI) that we implement using javax.swing. For the Controller, it is a part of data input that the source code is imported and data preparation for the Deadlock Detection Algorithm. Figure 24 shows the Class Diagram of the Deadlock Detection Tool. And the following sections elaborate more detail of each part.

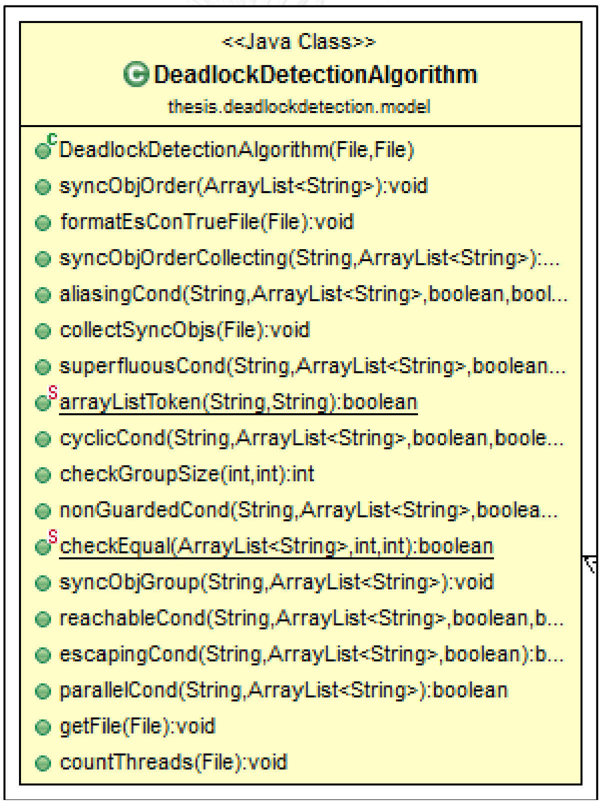


<<Java Class>>
DeadlockDetectionAlgorithm
thesis.deadlockdetection.model
DeadlockDetectionAlgorithm(File,File)
syncObjOrder(ArrayList<String>):void
formatEsConTrueFile(File):void
syncObjOrderCollecting(String,ArrayList<String>):...
aliasingCond(String,ArrayList<String>,boolean,bool...
collectSyncObjs(File):void
superfluousCond(String,ArrayList<String>,boolean...
arrayListToken(String,String):boolean
cyclicCond(String,ArrayList<String>,boolean,boole...
checkGroupSize(int,int):int
nonGuardedCond(String,ArrayList<String>,boolea...
checkEqual(ArrayList<String>,int,int):boolean
syncObjGroup(String,ArrayList<String>):void
reachableCond(String,ArrayList<String>,boolean,b...
escapingCond(String,ArrayList<String>,boolean):b...
parallelCond(String,ArrayList<String>):boolean
getFile(File):void
countThreads(File):void
<<Java Class>>
DeadlockOutputDisplay
thesis.deadlockdetection.view
DeadlockOutputDisplay(DefaultListModel<String>)
processGetFileInfo(DefaultListModel<String>):void
~display
0..1
<<Java Class>>
DeadlockDetectionTool
thesis.deadlockdetection.view
frmDeadlock: JFrame
textField: JTextField
textField_1: JTextField
pBarFrame: JFrame
pBar: JProgressBar
textField_2: JTextField
btnRun: JButton
dListModel: DefaultListModel<String>
list: JList<String>
textArea: JTextArea
main(String[]):void
DeadlockDetectionTool()
initialize():void
-task
0..1
<<Java Class>>
Task
thesis.deadlockdetection.view
Task()
doInBackground():Void
done():void
<<Java Class>>
DataPreparation
thesis.deadlockdetection.controller
sourceCodeFiles: ArrayList<File>
exportedDir: String
DataPreparation(File)
setExportedDir(String):void
getExportedDir():String
exportFiles(ArrayList<File>):void
getFiles(File[]):void

Figure 24 The Class Diagram of the Deadlock Detection Tool

3.2.1. Model Design

For the model, we implement the Deadlock Detection Algorithm. We create a package name is the thesis.deadlockdetection.model and the Class name is DeadlockDetectionAlgorithm.java. In Figure 24, the DeadlockDetectionAlgorithm receives input path of the source code of the multithreading API that already filtered unrelated source code from the DataPreparation Class. And the Deadlock Detection Algorithm processes to detect deadlock in the source code of the multithreading API when the Task is called from the View. Figure 25 shows the Class Diagram of the Deadlock Detection Algorithm.



<<Java Class>>
DeadlockDetectionAlgorithm
thesis.deadlockdetection.model
DeadlockDetectionAlgorithm(File,File)
syncObjOrder(ArrayList<String>):void
formatEsConTrueFile(File):void
syncObjOrderCollecting(String,ArrayList<String>):...
aliasingCond(String,ArrayList<String>,boolean,bool...
collectSyncObjs(File):void
superfluousCond(String,ArrayList<String>,boolean...
arrayListToken(String,String):boolean
cyclicCond(String,ArrayList<String>,boolean,boole...
checkGroupSize(int,int):int
nonGuardedCond(String,ArrayList<String>,boolea...
checkEqual(ArrayList<String>,int,int):boolean
syncObjGroup(String,ArrayList<String>):void
reachableCond(String,ArrayList<String>,boolean,b...
escapingCond(String,ArrayList<String>,boolean):b...
parallelCond(String,ArrayList<String>):boolean
getFile(File):void
countThreads(File):void

Figure 25 The Class Diagram of the Deadlock Detection Algorithm Class

3.2.2. View Design

For the View, we create a package name is the thesis.deadlockdetection.view. Inside the View package, we have the DeadlockDetectionTool.java that is implemented as a GUI of the tool and the DeadlockOutputDisplay.java that we implement the source code to show the result of the Deadlock Detection Tool. We implement the View using the javax.swing package. The DeadlockDetectionTool class gets input of the multithreading API path from a Browse button and shows the selected path in the text field. There is a Run button to perform deadlock detection for the multithreading API. When the tool is executing, there is a progress popup shows the percentage of the progress of the tool execution. The result of the deadlock shows in the Deadlock Result section. The Deadlock Detection Tool shows result that are the number of files that can detect deadlock, a list of files that have deadlock and the detail of deadlock from selected file. The detail of the deadlock shows when selecting a file name. The detail of the deadlock is shown in the text area that are the absolute path of the file, the line of code that shows deadlock sites, the number of threads that are created in the file and types of deadlock conditions. Figure 26 shows the screenshot of the GUI of the Deadlock Detection Tool.

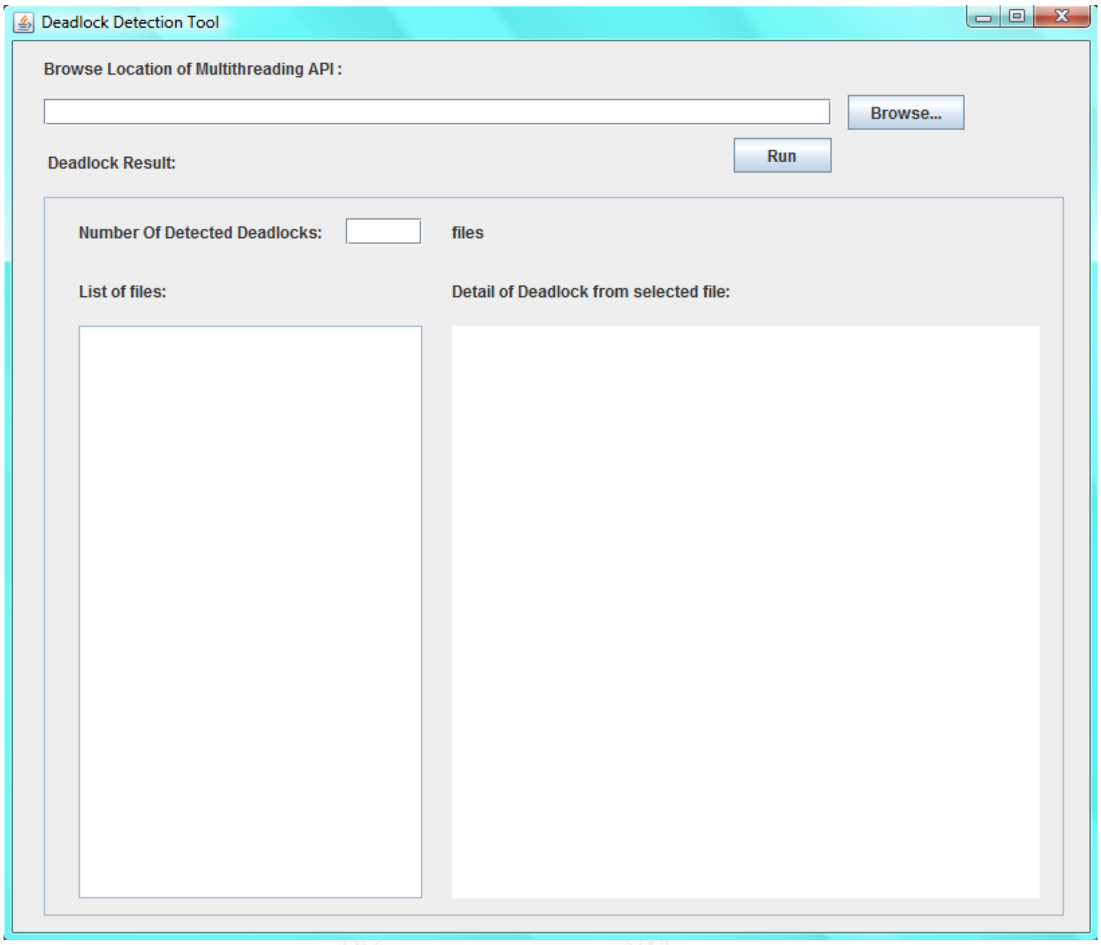

Deadlock Detection Tool
Browse Location of Multithreading API :
Browse...
Deadlock Result:
Run
Number Of Detected Deadlocks:
files
List of files:
Detail of Deadlock from selected file:

Figure 26 Graphic User Interface (GUI) of the Deadlock Detection Tool

Figure 24 the DeadlockDetectionTool Class that is the GUI of the Deadlock Detection Tool receives the input; the multithreading API path. After a user clicks the Run button, the Task for Deadlock Detection executes. The Task creates the DataPreparation Object, DeadlockDetectionAlgorithm Object and DeadlockOutputDisplay Object.

For the part of detail of Deadlock from selected file in Figure 26, we implement a Class named DeadlockOutputDisplay.java to get result from the Deadlock Detection Algorithm to display on the Text Area for the absolute path of the java file, lines of Deadlock site that occurs deadlock, the numbers of Thead that cause the deadlock and the types of deadlock conditions.

3.2.3. Controller Design

For the Controller, we create a package name is the thesis.deadlockdetection.controller. Inside the Controller package we have the DataPreparation.java. For preparing data, the tool gets the multithreading API source path from the GUI and it gets all java files of the multithreading API and then exports java files by filtering unrelated source code out. We are interested in the Class that extends the java.lang.Thread class and/or implements the java.lang.Runnable interface, thread.start(), synchronization methods, synchronization objects, wait() methods, notify() methods. The tool reads through all of files of source code of the multithreading API to get keywords that relates to deadlock that are “extends Thread”, “implements Runnable”, “synchronized”, “start()”, “wait()”, “notify()”, “notifyAll()”. The exported source code files are located in the new directory for the Deadlock Detection Algorithm Class to process deadlock detection. Figure 27 shows the Class Diagram of the DataPreparation.java.

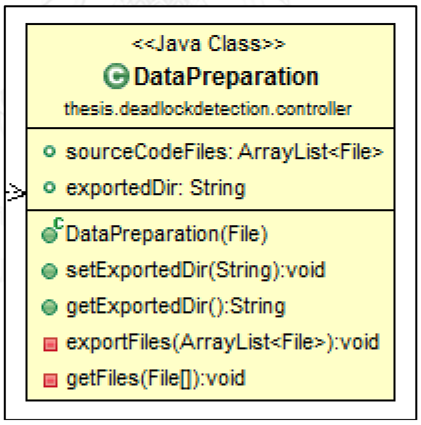

<<Java Class>>
DataPreparation
thesis.deadlockdetection.controller
sourceCodeFiles: ArrayList<File>
exportedDir: String
DataPreparation(File)
setExportedDir(String):void
getExportedDir():String
exportFiles(ArrayList<File>):void
getFiles(File[]):void

Figure 27 The Class Diagram of the DataPreparation Class

CHAPTER IV
RESULT AND VALIDATION

In this section, we provide the result of the Deadlock Detection Algorithm, the Deadlock Detection Tool and validation of the result.

4.1. Result

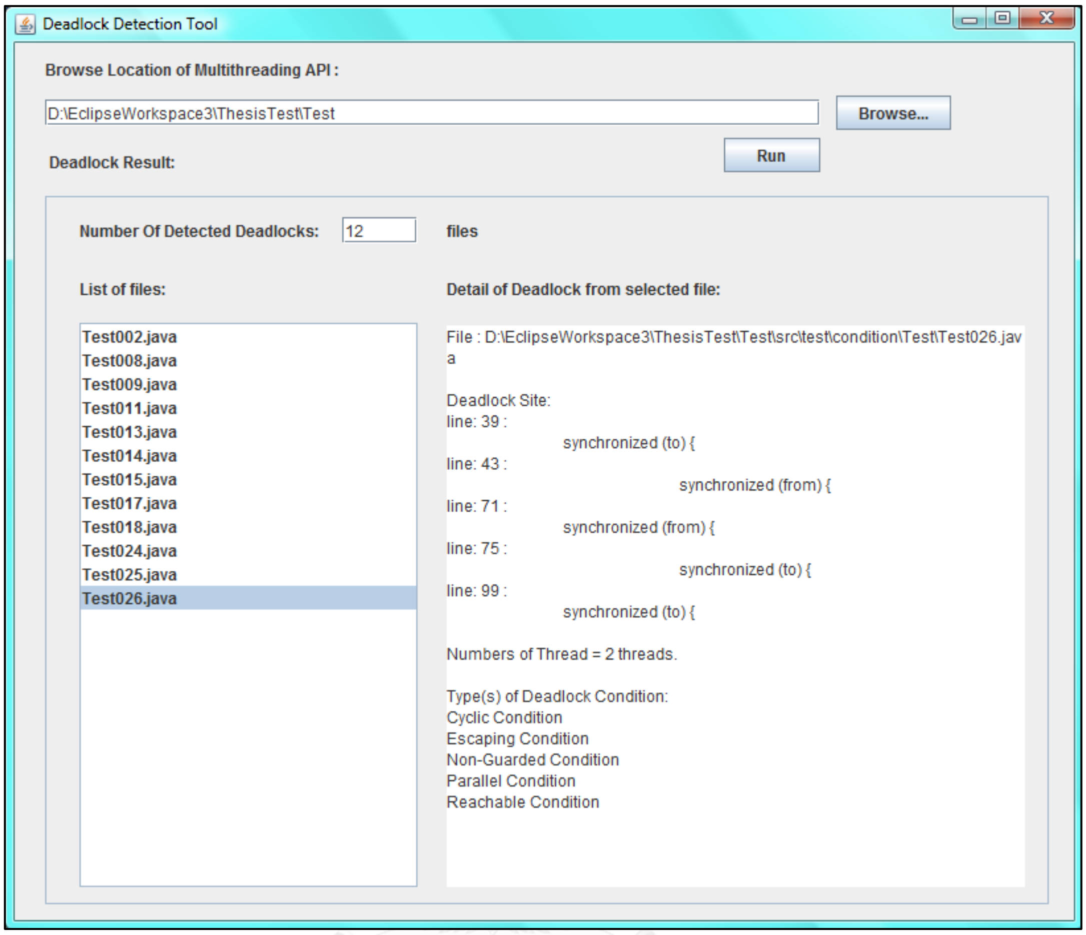
We have successfully implemented the Deadlock Detection Algorithm and the prototype of the Deadlock Detection Tool. The source code of each part is in the APPENDIX B and the screenshot of result of the Deadlock Detection Tool is in the APPENDIX C. We explain in detail of the result of the Deadlock Detection Algorithm and the Deadlock Detection Tool in the following sections.

4.4.1. Result of the Deadlock Detection Algorithm

Our Deadlock Detection Algorithm includes 7 deadlock conditions that are the Aliasing Condition, the Parallel Condition, the Escaping Condition, the Reachable Condition, the Superfluous Lock Condition, the Non-Guarded Lock Condition and the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition that are able to use in the Deadlock Detection Tool and they are able to provide the deadlock conditions and detect deadlock correctly.

4.4.2. Result of the Deadlock Detection Tool

To verify the Deadlock Detection Algorithm, we develop the prototype of the Deadlock Detection Tool to demonstrate using of the Deadlock Detection Algorithm to detect deadlock in the Java source code. The Deadlock Detection Tool can read the source code of the Java files or Java programs. We demonstrate a Java file as a source code of a multithreading API. Each Java file in a specified directory is an input of the Deadlock Detection Tool. The Deadlock Detection Tool provides the result to the GUI screen and log files. Figure 28 shows the result of the Deadlock Detection Tool that reports deadlock detection.


Deadlock Detection Tool
Browse Location of Multithreading API :
Browse...
Run
Deadlock Result:
List of files:
Detail of Deadlock from selected file:

D:\EclipseWorkspace3\ThesisTest\Test

Number Of Detected Deadlocks: 12 files

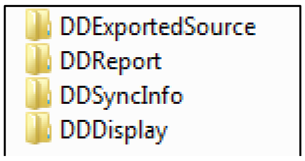
Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Test\Test026.jav
a

Deadlock Site:
line: 39 :
            synchronized (to) {
line: 43 :
                        synchronized (from) {
line: 71 :
            synchronized (from) {
line: 75 :
                        synchronized (to) {
line: 99 :
            synchronized (to) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 28 The output screenshot of the Deadlock Detection Tool that reports the result of deadlock detection

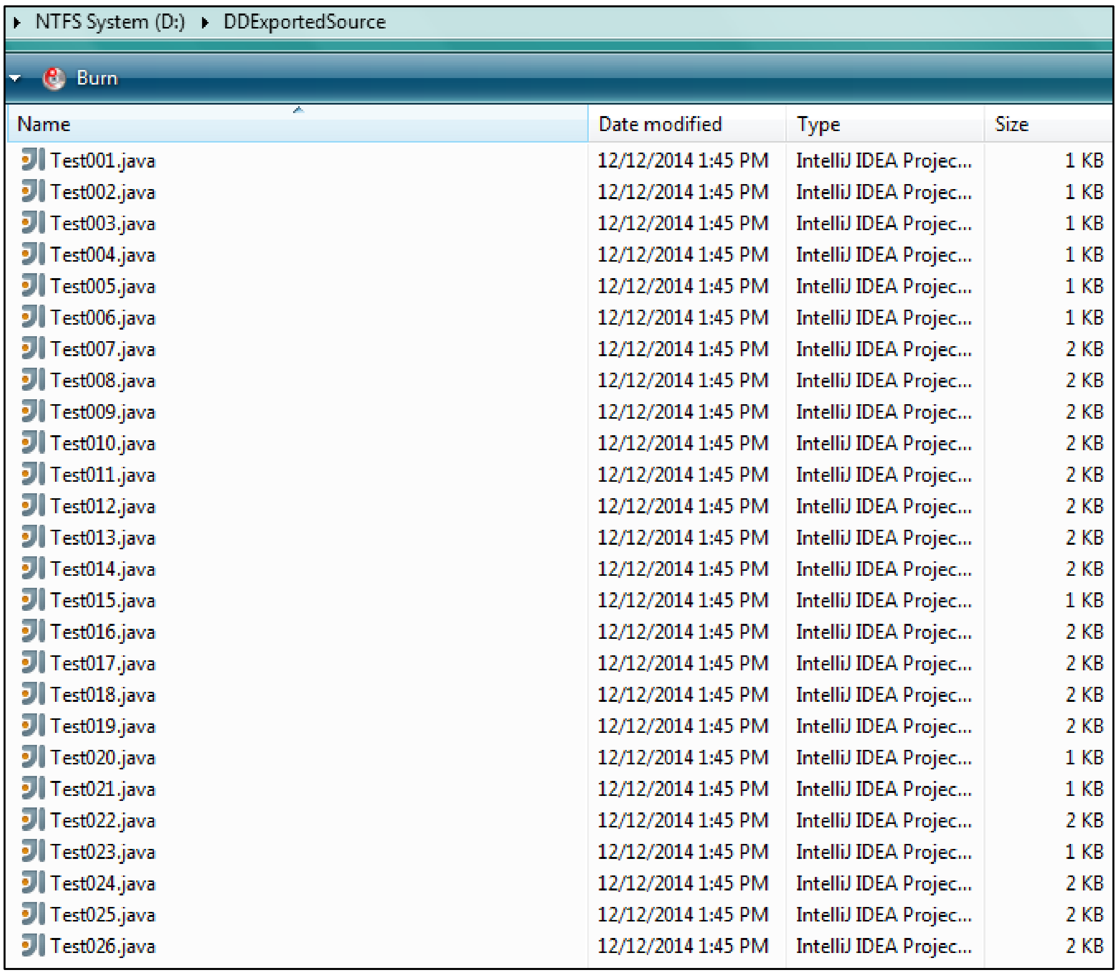
The Deadlock Detection Tool provides deadlock result that are the number of files that the Tool can detect deadlock in the text field, list of file names that have deadlock and the deteail of deadlock from the selected file in the text area. When user clicks a file name in the list, the detail of deadlock of that file is displayed in the text area. The detail that are the absolute path of the file, deadlock site that shows line number and lines of code of the Java file, number of Thread and the deadlock conditions that can detect the deadlock in the file.

The Deadlock Detection Tool also generates log files to use as a data for displaying on the screen and use in the Deadlock Detection Algorithm. The Deadlock Detection Tool generates the DDExportedSource directory to collect exported files of the Java source code that filters unrelated-source code out, a DDReport directory for collecting files of file report for each deadlock condition, a DDSyncInfo directory for collecting sequence of synchronized Objects of each files and a DDDisplay directory for collecting the displaying result of the file detail that have deadlock. Figure 29 shows the generated directories that produced by the Deadlock Detection Tool.



DDExportedSource
DDReport
DDSyncInfo
DDDisplay

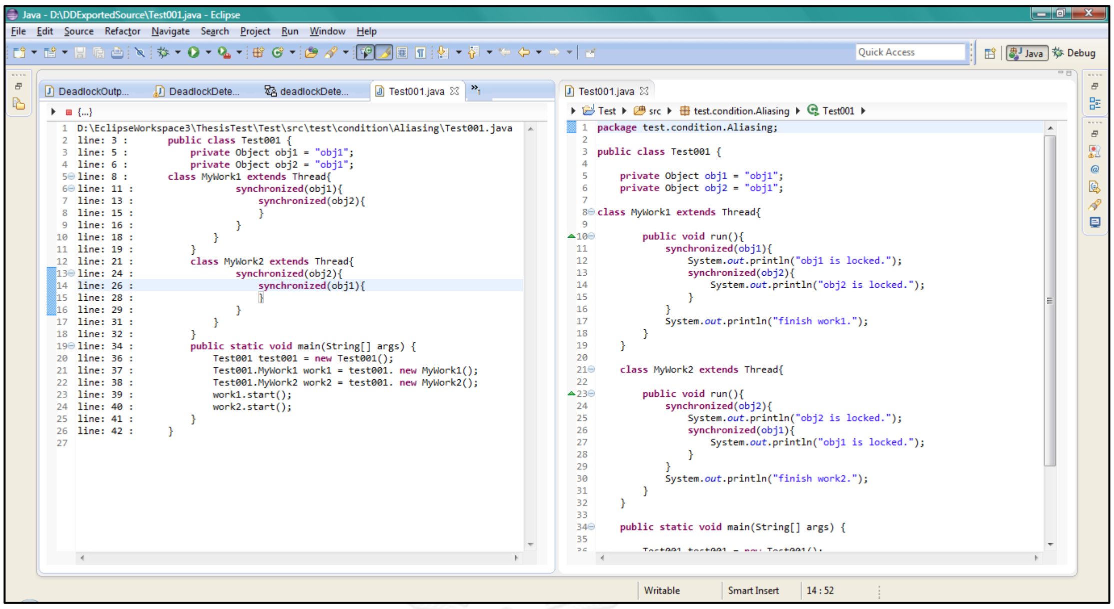
Figure 29 The directories that are the result of the Deadlock Detection Tool



Name	Date modified	Type	Size
Test001.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test002.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test003.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test004.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test005.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test006.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test007.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test008.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test009.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test010.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test011.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test012.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test013.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test014.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test015.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test016.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test017.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test018.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test019.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test020.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test021.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test022.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test023.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	1 KB
Test024.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test025.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB
Test026.java	12/12/2014 1:45 PM	IntelliJ IDEA Projec...	2 KB

Figure 30 The DDExportedSource Directory

Figure 30 shows inside of the DDExportedSource directory that it has exported files that already filtere out unrelated lines of code.

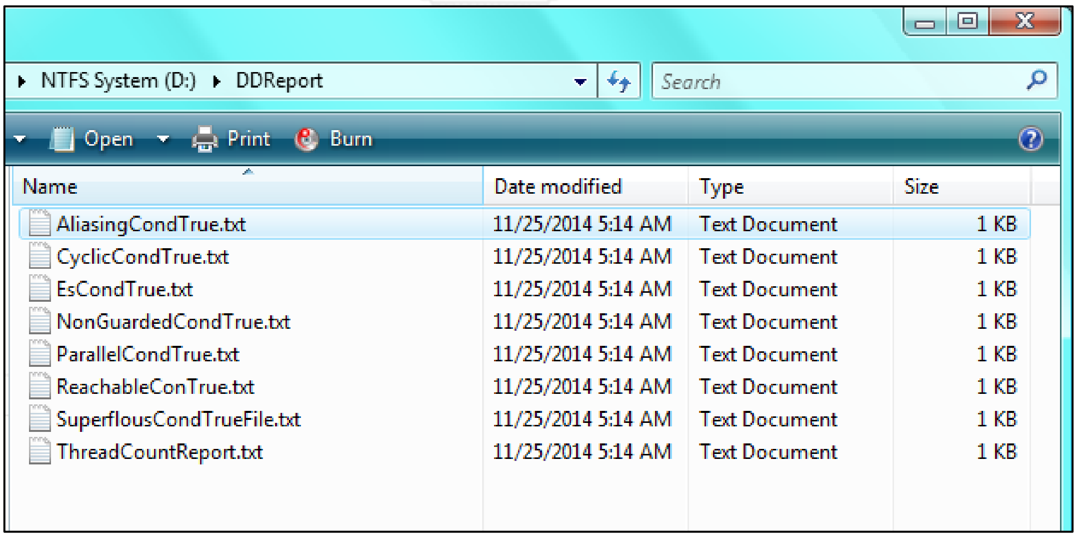

Java - D:\DDExportedSource\Test001.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
DeadlockOutp...
DeadlockDete...
deadlockDete...
Test001.java
Quick Access
Java
Debug
Test ▸ src ▸ test.condition.Aliasing ▸ Test001
Writable
Smart Insert
14 : 52

```
D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Aliasing\Test001.java
line: 3 :       public class Test001 {
line: 5 :           private Object obj1 = "obj1";
line: 6 :           private Object obj2 = "obj1";
line: 8 :       class MyWork1 extends Thread{
line: 11 :                  synchronized(obj1){
line: 13 :                      synchronized(obj2){
line: 15 :                      }
line: 16 :                  }
line: 18 :              }
line: 19 :          }
line: 21 :          class MyWork2 extends Thread{
line: 24 :                  synchronized(obj2){
line: 26 :                      synchronized(obj1){
line: 28 :                      }
line: 29 :                  }
line: 31 :              }
line: 32 :          }
line: 34 :          public static void main(String[] args) {
line: 36 :              Test001 test001 = new Test001();
line: 37 :              Test001.MyWork1 work1 = test001. new MyWork1();
line: 38 :              Test001.MyWork2 work2 = test001. new MyWork2();
line: 39 :              work1.start();
line: 40 :              work2.start();
line: 41 :          }
line: 42 :      }
```

```
package test.condition.Aliasing;

public class Test001 {

    private Object obj1 = "obj1";
    private Object obj2 = "obj1";

class MyWork1 extends Thread{

        public void run(){
            synchronized(obj1){
                System.out.println("obj1 is locked.");
                synchronized(obj2){
                    System.out.println("obj2 is locked.");
                }
            }
            System.out.println("finish work1.");
        }
    }

    class MyWork2 extends Thread{

        public void run(){
            synchronized(obj2){
                System.out.println("obj2 is locked.");
                synchronized(obj1){
                    System.out.println("obj1 is locked.");
                }
            }
            System.out.println("finish work2.");
        }
    }

    public static void main(String[] args) {

```

Figure 31 The exported source code compares with the original source code

Figure 31 shows the exported source code compares with the original source code. The exported source code is added line number and filtered out unrelated lines of code.

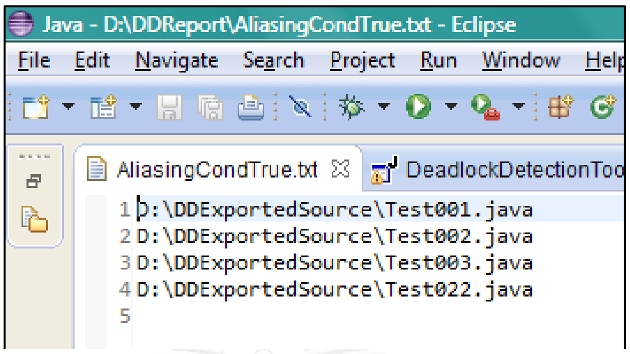


Name	Date modified	Type	Size
AliasingCondTrue.txt	11/25/2014 5:14 AM	Text Document	1 KB
CyclicCondTrue.txt	11/25/2014 5:14 AM	Text Document	1 KB
EsCondTrue.txt	11/25/2014 5:14 AM	Text Document	1 KB
NonGuardedCondTrue.txt	11/25/2014 5:14 AM	Text Document	1 KB
ParallelCondTrue.txt	11/25/2014 5:14 AM	Text Document	1 KB
ReachableConTrue.txt	11/25/2014 5:14 AM	Text Document	1 KB
SuperflousCondTrueFile.txt	11/25/2014 5:14 AM	Text Document	1 KB
ThreadCountReport.txt	11/25/2014 5:14 AM	Text Document	1 KB

Figure 32 The DDReport directory

Figure 32 shows the DDReport directory that has files of the result of each condition that are the Aliasing Condition (AliasingCondTrue.txt), the Cyclic Lock Dependency Condition (CyclicCondTrue.txt), the Escaping Condition (EsConTrue.txt), the Non-Guarded Lock Condition (NonGuardedCondTrue.txt), the Parallel Condition

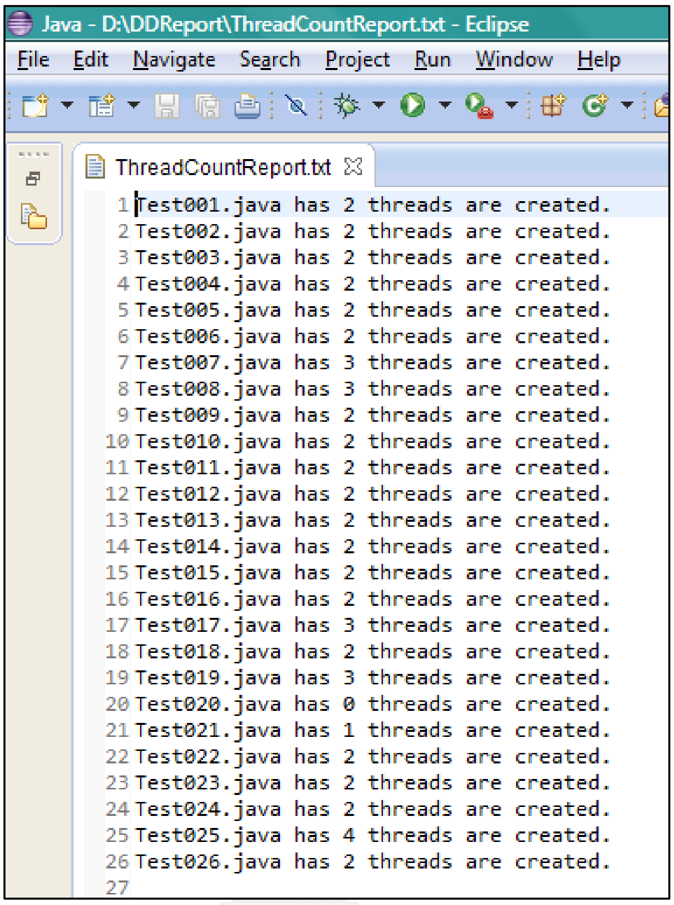
(ParallelCondTrue.txt), the Reachable Condition (ReachableCondTrue.txt), the Superfluous Lock Condition (SuperfluousCondTrue.txt) and Thread Count Report (ThreadCountReport.txt). In the files of condition result contains the list of file names that returns true for that condition.


Java - D:\DDReport\AliasingCondTrue.txt - Eclipse
File Edit Navigate Search Project Run Window Help
AliasingCondTrue.txt
DeadlockDetectionToo

```
D:\DDExportedSource\Test001.java
D:\DDExportedSource\Test002.java
D:\DDExportedSource\Test003.java
D:\DDExportedSource\Test022.java
```

Figure 33 The content of the AliasingCondTrue.txt

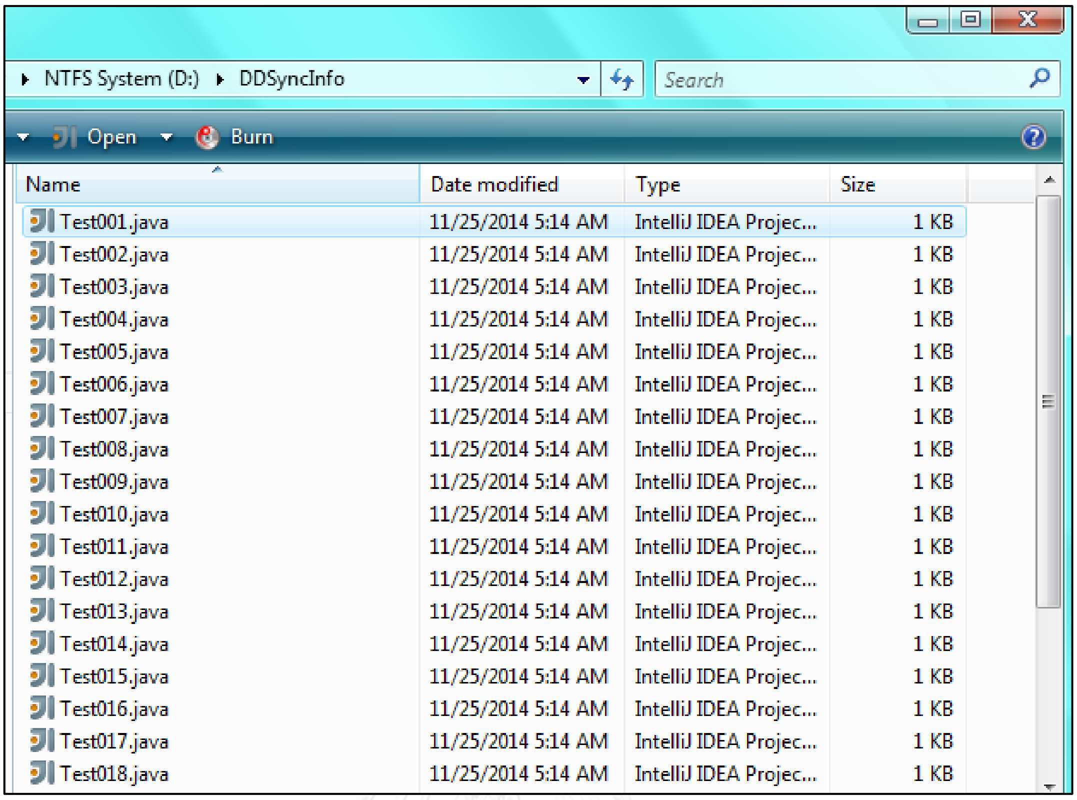
Figure 33 shows content of the AliasingCondTrue.txt that is a file in the DDReport directory. The AliasingCondTrue.txt has the list of file names that the Deadlock Detection Algorithm can detect that files are the Aliasing condition.


Java - D:\DDReport\ThreadCountReport.txt - Eclipse
File Edit Navigate Search Project Run Window Help
ThreadCountReport.txt

```
Test001.java has 2 threads are created.
Test002.java has 2 threads are created.
Test003.java has 2 threads are created.
Test004.java has 2 threads are created.
Test005.java has 2 threads are created.
Test006.java has 2 threads are created.
Test007.java has 3 threads are created.
Test008.java has 3 threads are created.
Test009.java has 2 threads are created.
Test010.java has 2 threads are created.
Test011.java has 2 threads are created.
Test012.java has 2 threads are created.
Test013.java has 2 threads are created.
Test014.java has 2 threads are created.
Test015.java has 2 threads are created.
Test016.java has 2 threads are created.
Test017.java has 3 threads are created.
Test018.java has 2 threads are created.
Test019.java has 3 threads are created.
Test020.java has 0 threads are created.
Test021.java has 1 threads are created.
Test022.java has 2 threads are created.
Test023.java has 2 threads are created.
Test024.java has 2 threads are created.
Test025.java has 4 threads are created.
Test026.java has 2 threads are created.
```

Figure 34 The ThreadCountReport.txt

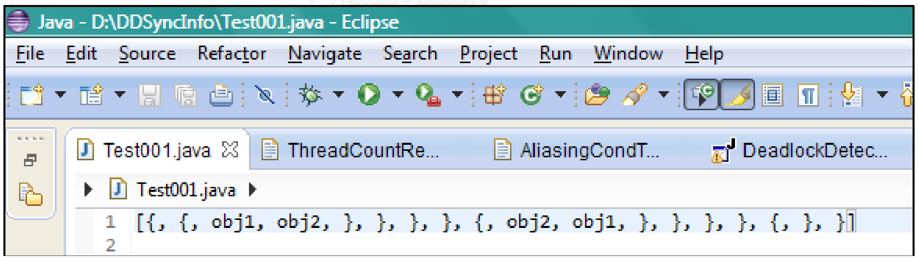
Figure 34 shows the content in the ThreadCountReport.txt file that has the result of the Deadlock Detection Algorithm for the number of Thread of each file.



NTFS System (D:) ▸ DDSyncInfo

Name	Date modified	Type	Size
Test001.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test002.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test003.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test004.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test005.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test006.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test007.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test008.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test009.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test010.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test011.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test012.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test013.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test014.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test015.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test016.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test017.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB
Test018.java	11/25/2014 5:14 AM	IntelliJ IDEA Projec...	1 KB

Figure 35 The DDSyncInfo directory

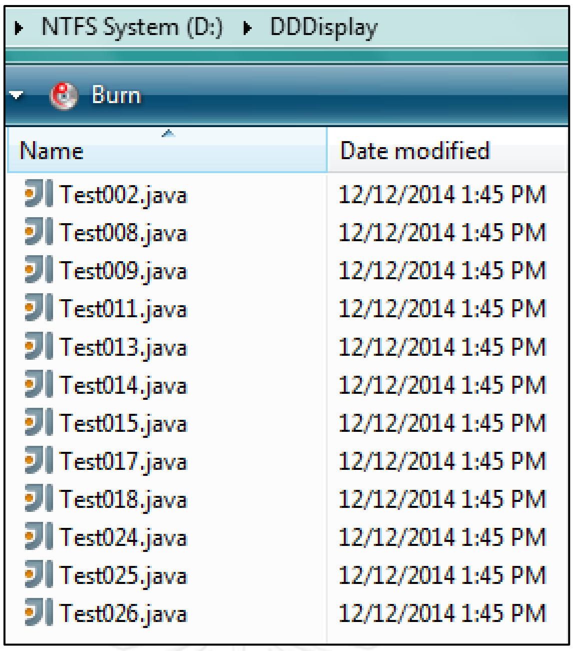


Java - D:\DDSyncInfo\Test001.java - Eclipse

```
[{, {, obj1, obj2, }, }, }, }, {, obj2, obj1, }, }, }, }, {, }, }]
```

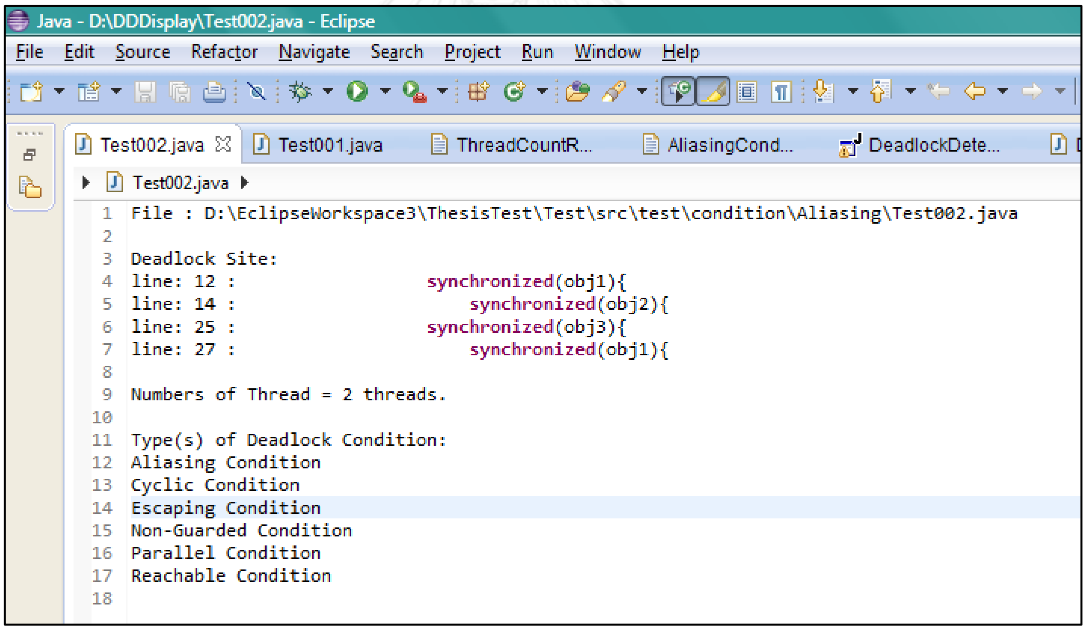
Figure 36 The result of the file in the DDSyncInfo directory

Figure 35 shows the list of files in the DDSyncInfo directory. Figure 36 shows the content in one of file in the DDSyncInfo directory. In the file, it has the sequence of the synchronized Objects of the imported Java file.



Name	Date modified
Test002.java	12/12/2014 1:45 PM
Test008.java	12/12/2014 1:45 PM
Test009.java	12/12/2014 1:45 PM
Test011.java	12/12/2014 1:45 PM
Test013.java	12/12/2014 1:45 PM
Test014.java	12/12/2014 1:45 PM
Test015.java	12/12/2014 1:45 PM
Test017.java	12/12/2014 1:45 PM
Test018.java	12/12/2014 1:45 PM
Test024.java	12/12/2014 1:45 PM
Test025.java	12/12/2014 1:45 PM
Test026.java	12/12/2014 1:45 PM

Figure 37 The DDDisplay directory


Java - D:\DDDisplay\Test002.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Test002.java
Test001.java
ThreadCountR...
AliasingCond...
DeadlockDete...
Test002.java

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Aliasing\Test002.java

Deadlock Site:
line: 12 :                  synchronized(obj1){
line: 14 :                      synchronized(obj2){
line: 25 :                  synchronized(obj3){
line: 27 :                      synchronized(obj1){

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Aliasing Condition
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition

```

Figure 38 The content of the file in the DDDisplay directory

Figure 37 shows the list of file inside the DDDisplay directory. Figure 38 shows the content of the file in the DDDisplay directory. The content is the result of the selected file that shows on the text area when user selects the file to see the detail of the result of the deadlock detection.

4.2. Validation

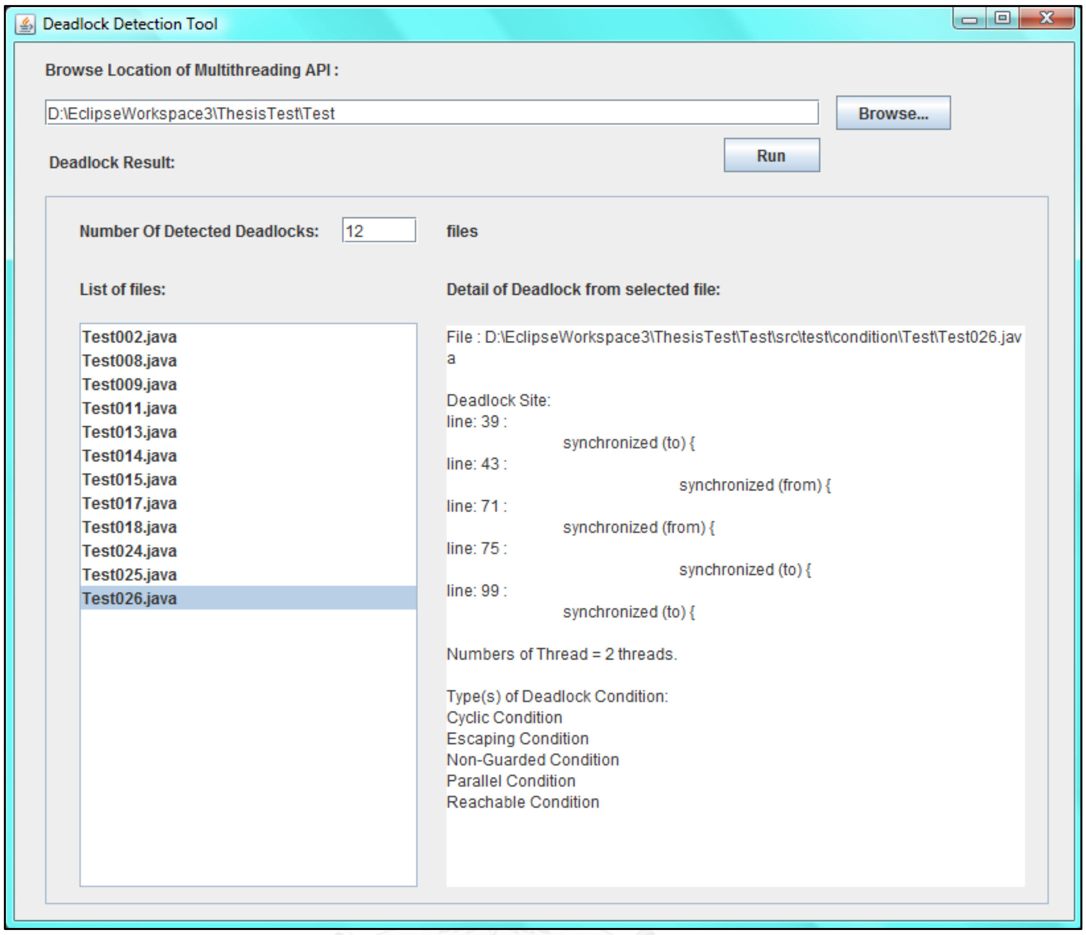
To validate the result of the Deadlock Detection Algorithm, we create test files to test the 7 deadlock conditions and deadlock. We create 26 test files. All test files are implemented using the synchronization best practices [21] that suggests minimizing the number of Threads and synchronization. The nested synchronization should not have more than 2 or 3. Therefore our test files have 2 or 3 Threads and 2 or 3 nested synchronization. The detail of all test files is added in APPENDIX D. Table 13 shows the test result of running of the Deadlock Detection Algorithm for 26 test files comparing between the expeted results and the actual results.

Table 13 The result of the Deadlock Detection Tool when using test files for testing

Files (.java)	Result	Dead lock	Aliasing	Cyclic Lock Dependency	Escaping	Parallel	Non-Guarded Lock	Reachable	Superfluous Lock	Thread Number
Test001	Expected	N	x		x	x	x	x	x	2
Test001	Actual	N	x		x	x	x	x	x	2
Test002	Expected	Y	x	x	x	x	x	x		2
Test002	Actual	Y	x	x	x	x	x	x		2
Test003	Expected	N	x		x	x		x		2
Test003	Actual	N	x		x	x		x		2
Test004	Expected	N			x	x	x	x		2
Test004	Actual	N			x	x	x	x		2
Test005	Expected	N			x	x	x	x		2
Test005	Actual	N			x	x	x	x		2
Test006	Expected	N			x	x	x	x		2
Test006	Actual	N			x	x	x	x		2
Test007	Expected	N			x	x	x	x		3
Test007	Actual	N			x	x	x	x		3
Test008	Expected	Y		x	x	x	x	x		3
Test008	Actual	Y		x	x	x	x	x		3
Test009	Expected	Y		x	x	x	x	x		2
Test009	Actual	Y		x	x	x	x	x		2
Test010	Expected	N			x	x		x		2
Test010	Actual	N			x	x		x		2
Test011	Expected	Y		x	x	x	x	x		2
Test011	Actual	Y		x	x	x	x	x		2
Test012	Expected	N			x	x		x		2
Test012	Actual	N			x	x		x		2
Test013	Expected	Y		x	x	x	x	x		2

Files (.java)	Result	Dead lock	Aliasing	Cyclic Lock Dependency	Escaping	Parallel	Non-Guarded Lock	Reachable	Superfluous Lock	Thread Number
Test013	Actual	Y		x	x	x	x	x		2
Test014	Expected	Y		x	x	x	x	x		2
Test014	Actual	Y		x	x	x	x	x		2
Test015	Expected	Y		x	x	x	x	x		2
Test015	Actual	Y		x	x	x	x	x		2
Test016	Expected	N			x	x		x		2
Test016	Actual	N			x	x		x		2
Test017	Expected	Y		x	x	x	x	x		3
Test017	Actual	Y		x	x	x	x	x		3
Test018	Expected	Y		x	x	x	x	x		2
Test018	Actual	Y		x	x	x	x	x		2
Test019	Expected	N			x	x		x		3
Test019	Actual	N			x	x		x		3
Test020	Expected	N								0
Test020	Actual	N								0
Test021	Expected	N					x			1
Test021	Actual	N					x			1
Test022	Expected	N	x		x	x	x	x		2
Test022	Actual	N	x		x	x	x	x		2
Test023	Expected	N	x		x	x	x			2
Test023	Actual	N	x		x	x	x	x		2
Test024	Expected	Y		x	x	x	x	x		2
Test024	Actual	Y		x	x	x	x	x		2
Test025	Expected	Y		x	x	x	x	x		4
Test025	Actual	Y		x	x	x	x	x		4
Test026	Expected	Y		x	x	x	x	x		2
Test026	Actual	Y		x	x	x	x	x		2

Table 13 shows that all actual results of testing are the same as expected results. The Deadlock Detection Algorithm provides correct results. The result is 100 percentage of correctness. The Deadlock Detection Tool provides a result with 12 files that have deadlock that are Test002.java, Test008.java, Test009.java, Test011.java, Test013.java, Test014.java, Test015.java, Test017.java, Test018.java, Test024.java, Test025.java and Test026.java. The results are correct as expected. Figure 39 shows the screenshot of the Deadlock Detection Tool that provides correct result of deadlock detection.


Deadlock Detection Tool
Browse Location of Multithreading API :
Browse...
Run
Deadlock Result:
Number Of Detected Deadlocks:
files
List of files:
Detail of Deadlock from selected file:

D:\EclipseWorkspace3\ThesisTest\Test

Number Of Detected Deadlocks: 12 files

Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Test\Test026.jav
a

Deadlock Site:
line: 39 :
                synchronized (to) {
line: 43 :
                                synchronized (from) {
line: 71 :
                synchronized (from) {
line: 75 :
                                synchronized (to) {
line: 99 :
                synchronized (to) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 39 the screenshot of the Deadlock Detection Tool that provides correct result of deadlock detection

CHAPTER V
CONCLUSION AND FUTURE WORK

We successfully develop the Deadlock Detection Algorithm to detect the deadlock in the multithreading API. We implement the prototype of the Deadlock Detection Tool to demonstrate using of the Deadlock Detection Algorithm and it provides correct result as expected. The Deadlock Detection Algorithm can detect the deadlock correctly by analyzing the source code for the deadlock conditions that are the Aliasing Condition, the Escaping Condition, the Parallel Condition, the Non-Guarded Lock Condition, the Reachable Condition, the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition and the Superfluous Lock Condition.

The meaning of each deadlock conditions is explained as follows:

1. Parallel Condition

The Parallel Condition is the condition that there are several Threads are created and run simultaneously and each Thread locks Objects. For example; in the program there are 2 Threads are created and run simultaneously that are Thread A and Thread B. Thread A locks Object L1 and then locks Object L2 sequentially. Thread B locks Object L3.

2. Escaping Condition

The Escaping Condition is the condition that a lock can be accessible from more than one thread. For example; there is Object L1, it can be accessible by Thread A and Thread B.

3. Reachable Condition

The Reachable Condition is the condition that a Thread can reach to a lock and acquire it and then reach to another lock and is still holding the first lock. For example; Thread A reaches to Object L1 and acquires Object L1 and then it reaches to Object L2. Thread A is still holding Object L1 when reaches to Object L2.

4. Aliasing Condition

The Aliasing Condition of the Deadlock Detection Algorithm is the condition that a lock and another lock are referring to the same Object. For example; if Thread A acquires Object L1 and then acquires Object L2. Object L1 is a reference of Object L2. Object L1 and Object L2 is the aliasing objects.

5. Superfluous Lock Condition

The Superfluous Lock Condition is the condition that the synchronized Objects are unnecessary duplicate locked. For example; Thread A acquires Object L1 and after that in the source code Thread A calls to acquires Object L1 again. Thread A acquires Object L1 twice that is unnecessary duplicate lock. The Superfluous Lock can cause the system unnecessary block an Object and lead to deadlock.

6. Non-Guarded Lock Condition

The Non-Guarded Lock Condition is the invert condition of the Guarded Lock Condition. The Guarded Condition helps to solve deadlock problems by adding the same synchronized Object before the couple of reverse order of synchronized Objects that are locked by threads. The synchronized Object is called the Guarded Lock or the common lock. The Guarded Lock prevents deadlock occurrence. When the Thread acquires the Guarded Lock, other thread cannot hold that Guarded Lock until the Thread releases the Guarded Lock. Therefore the reverse order of synchronized Objects of other threads cannot access until the first thread that acquires the guarded lock finishes the task and releases the Guarded Lock. For example; if there are Thread A and Thread D. Thread A has acquired sequence of Objects that are L4, L1 and L2 and Thread B has acquired sequence of Objects that are L4, L2 and L1.

7. Reverse Order Lock Conditon or Cyclic Lock Dependency Condition

The Reverse Order Lock Condition or Cyclic Lock Dependency Condition is a condition that the synchronized Objects locked by two or more than two threads are reverse. When displaying on the Wait-For Graph it shows cyclic lock order. For example; Thread A acquires L1 and L2 and Thread D acquires L2 and L1 sequentially. The lock order of Thread A and Thread D is reverse. It causes deadlock. When Thread A owns L1 and Thread D owns L2, after that Thread A waits to acquire L2 but L2 is owned by Thread D that waits to acquire L1 that is owned by Thread A. Both threads lock Objects in reverse order.

The sequence of the Deadlock Detection Algorithm for detecting deadlock is Thread counting, Synchronized Object collecting, the Aliasing Condition, the Parallel Condition, the Escaping Condition, the Reachable Condition, the Superfluous Lock Condition, the Non-Guarded Lock Condition and the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition.

For the implementation of the Deadlock Detection Algorithm, Thread Counting checks how many thread are created in the source code by counting the line of the source code that calls the start() method. The Class that extends the Thread Class and/or implements the Runnable Class is able to call start() method to create a Thread to perform a task. After that the algorithm collects synchronized Objects from the line of source code that has “synchronized”. The synchronized Objects are collected in the ArrayList as regards their sequence in the source code. The Deadlock Detection Algorithm uses the ArrayList of the synchronized Objects to check the deadlock conditions and analyzes the deadlock. The next process is the Aliasing Condition. The algorithm checks whether there are aliasing Objects in the source code or not, if there are aliasing of Objects in the source code, the synchronized Objects in the ArrayList are replaced with their aliasing. After that the Deadlock Detection Algorithm processes the Parallel Condition. The Deadlock Detection Algorithm checks whether there are several threads created and lock Objects or not. If there is only one thread, the source code does not have deadlock. The Parallel Condition returns false and informs there is no deadlock occurs in the source code. If there are several threads created and lock Objects, the source code potentially encounters deadlock. The Parallel Condition returns true. And the Deadlock Detection Algorithm processes to the next condition that is the Escaping Condition. The Escaping Condition is the condition that a lock can be accessible from more than one thread. Therefore the Deadlock Detection Algorithm checkes whether each synchronized Object in the ArrayList is duplicate or not. If it is duplicate, it means that there are several threads acquire to lock this Object. We use the HashMap to check the duplication. The synchronized Object is added as a key for the HashMap and the number of lock that Object is value. If the value is equal 0, the Object is not duplicate. The Escaping Condition returns false and the Deadlock Detection Algorithm breaks and informs deadlock does not occur. If value more than 0, it is duplicate and the Escaping Condition returns true. The source code has potentially that deadlock occurs. The Deadlock Detection Algorithm processes the next condition that is the Reachable Condition. For the Reachable Condition, the Deadlock Detection Algorithm checks whether the number of Thread more than 1 and each thread locks several Objects or not. If no, the Reachable Condition returns false, the Deadlock Detection Algorithm stops and deadlock does not occur. If yes, the Reachable Condition returns true and deadlock potentially occurs and the Deadlock Detection Algorithm processes to check the next condition that is the Superfluous Lock Condition. For the Superfluous Lock Condition, the Deadlock Detection Algorithm checks whether the adjacent Objects in the ArrayList of each thread are the same or not. If no, the Superfluous Lock Condition returns false and the Deadlock Detection Algorithm stops process and it informs that deadlock does not occur. If yes, the Superfluous Lock Condition returns true. Deadlock potentially occurs and the Deadlock Detection Algorithm processes the next condition that is the Non-Guarded Lock Condition. For the Non-Guarded Lock Condition, the Deadlock Detection Algorithm checks whether the first synchronized Object of each Thread is the same Object or not. If yes, it is the Guarded Lock Condition and deadlock does not occur. The Deadlock Detection Algorithm stops processing to the next condition. If no, it is the Non-Guarded Lock Condition, the deadlock potentially occurs and the Deadlock Detection Algorithm continues processing the next condition that is the Reverse Order Locking Conditon or the Cyclic Lock Dependency Condition. For the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition, the Deadlock Detection Algorithm checks whether the lock order is reverse or the cyclic lock occurs or not. If no, the source code does not have deadlock. If yes, the deadlock potentially occurs.

In conclusion, using five deadlock conditions [3] and two code patterns of deadlock conditions [15] that is enhanced from the necessary conditions of deadlock of Edward G. Coffman, Jr. [9] is able to develop an algorithm to detect deadlock in the multithreading API. We found that the Reverse Order Locking Condition or the Cyclic Lock Dependency Condition is the final result of the Deadlock Detection Algorithm that helps us to detect and predicts whether deadlock potentially occurs or not. In addition, the result from other conditions helps us to understand which deadlock conditions occur and relate to deadlock.

The found of this thesis can help the developer to check the source code whether the deadlock potentially occurs or not. They can perform checking the source code in the early phase of the Software Development or the Software Testing as a static analysis before the defect is detected in the application. Moreover, the developer is able to use information of the deadlock conditions to investigate and resolve the problem in the multithreading API. In addition, the developer is able to use the Deadlock Detection Algorithm for the unit test. Even if the developer cannot fix the problem, they can also use the result of deadlock detection to provide as a suggestion or guideline or document it for the usage and limitation of the multithreading API to customers or team. Furthermore, the result of deadlock detection can help the developer to understand Object synchronization in the multithreading API or source code. Last but not least it can help the development manager to prepare for issue that will happen in the future and to plan the resources for fixing the defect or to decide whether it is affect to the design or requires to fix or not.

For the limitation of this thesis, as regards we use the Oracle implementation best practice [21] that suggests to avoid and minimize creating several threads and synchronization and nested synchronization. Therefore our algorithm is able to detect deadlock only for the class that has 2 or 3 threads and 2 or 3 nested synchronization. In addition, we emphasize in developing the Deadlock Detection Algorithm as a major work of this thesis. The Deadlock Detection Tool is able to import the source code of Java programs. Each file of the Java source code is represented as a multithreading API and each file are not relate or calls other files. The program solely completes tasks in its file.

For the future work we can use the algorithm to detect the deadlock in the UML of the multithreading API by changing the input part of the Deadlock Detection Tool to read the UML instead. It will help to detect the deadlock in the early process of the Software Development Life Cycle to decrease the cost of fixing bugs. In addition, it is possible to add the Reentrance Condition to detect deadlock that occurs from java.util.concurrent.locks.ReentrantLock implementation. The algorithm should be able to detect the lock() method and the unlock() method [22]. Moreover, it is valuable to develop the algorithm and tool to support deadlock that more complex in the way of using more than 3 threads and more than 3 nested synchronization. It is possible to develop the Deadlock Detection Algorithm and Tool using the call graph theory for higher size of the source code of the multithreading API or the application. The call graph theory can help to implement the deadlock detection algorithm and tool using other static analysis methods and can help the developer to enhance visibility and understanding of the multithreading API behavior and detect deadlock.

REFERENCES

1. Pedro Fonseca, C.L., and Rodrigo Rodrigues, *Finding Complex Concurrency Bugs in Large Multi-Threaded Applications*, in *EuroSys'11*. 2011, ACM: Salzburg, Austria. p. 215-228.
2. Tong Li, C.S.E., Alvin R. Lebeck, and Daniel J. Sorin. *Pulse: A Dynamic Deadlock Detection MechanismUsing Speculative Execution*. in *2005 USENIX Annual Technical Conference*. 2005. Anaheim, California.
3. Mayur Naik, C.-S.P., Koushik Sen and David Gay, *Effective Static Deadlock Detection*, in *ICSE'09*. 2009, IEEE: Vancouver, Canada. p. 386-396.
4. Holt, R.C., *Some Deadlock Properties of Computer Systems.* Computing Surveys, 1972. 4(3): p. 18.
5. Henning, M., *API design matters.* Communications of the ACM, 2009. 52(5): p. 46.
6. Shan Lu, S.P., Eunsoo Seo and Yuanyuan Zhou, *Learning from mistakes - a comprehensive study on real world concurrency bug characteristics*, in *ASPLOS'08*. 2008, ACM: Seattle, Washington, USA. p. 329-339.
7. affiliates, O.a.o.i. *About the Java Technology*. Oracle and/or its affiliates 1995, 2012 [cited 2012 19/10]; Available from: http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html.
8. affiliates, O.a.o.i. *Deadlock*. 1995, 2012 [cited 2012 22/09]; Available from: http://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html.
9. Edward G. Coffman, J., *Deadlock*, in *Wikipedia*. 1971.
10. Knapp, E., *Deadlock Detection in Distributed Databases.* ACM Computing Surveys, 1987. 19(4): p. 26.
11. Mattia Monga, R.P., Emanuele Passerini, *A Hybrid Analysis Framework For Detecting Web Application Vulnerabilities*, in *ICSE*. 2009, Software Engineering for Secure Systems, 2009. SESS '09. : Vancouver, Canada. p. 25 - 32.
12. Elmagarmid, A.K., *A Survey of Distributed Deadlock Detection Algorithms* SIGMOD RECORD, 1986. 15(3): p. 9.

13. Ismail, U., *Incremental Call Graph Construction for the Eclipse IDE*. 2009, David R. Cheriton School of Computer Science, University of Waterloo. p. 9.
14. Shujuan Jiang, Y.Z.a.D.Y., *Test Data Generation Approach for Basis Path Coverage.* ACM SIGSOFT Software Engineering Notes, 2012. 37(3): p. 7.
15. Frank Otto, T.M., *Finding Synchronization Defects in Java Programs Extended Static Analyses and Code Patterns*, in *IWMSE'08*. 2008, ACM: Leipzig, Germany. p. 41-46.
16. Jonas Trümper, J.B., Jürgen Döllner *Understanding Complex Multithreaded Software Systems by Using Trace Visualization*, in *SOFTVIS'10*. 2010, ACM: Salt Lake City, Utah, USA. p. 133-142.
17. Stoller, R.A.a.S.D., *Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables*, in *PADTAD-IV*. 2006, ACM: Portland, Maine, USA. p. 51-59.
18. Amy Williams, W.T., and Michael D. Ernst. *Static Deadlock Detection for Java Libraries*. in *ECOOP2005*. 2005. Glasgow UK.
19. Jyotirmoy Deshmukh, E.A.E.a.S.S. *Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients*. in *2009 IEEE/ACM International Conference on Automated Software Engineering*. 2009. Washington, DC, USA: IEEE Computer Society.
20. Gianpiero Francesca, A.S., Gigliola Vaglini and Maria Luisa Villani. *Ant Colony Optimization for Deadlock Detection in Concurrent Systems*. in *2011 35th IEEE Annual Computer Software and Applications Conference*. 2011. Washington, DC, USA: IEEE Computer Society.
21. Oracle. *Oracle9i Application Server Best Practices*. 2003 2003 [cited 2014 18 October]; Release 2 (9.0.3) Part Number B10578-02:[Avoid or Minimize Synchronization]. Available from: http://docs.oracle.com/cd/A97688_16/generic.903/bp/java.htm#1005570.
22. Oracle. *Package java.util.concurrent.locks*. 1993; Available from: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/package-summary.html.

APPENDICES

APPENDIX A
PUBLICATION

1. Ploysri, S., and Rivepiboon, W. A Development of the Deadlock Detection Algorithm using Static Analysis for Potential Deadlocks in the Multithreading API. The proceeding of the International Multi-Conferences 2013 (ICACCT 2013), International Conference on Advances in Computing and Communication Technologies, 2013: 9-15.
2. Ploysri, S., and Rivepiboon, W. Yet, Another Method for Detecting API Deadlock. The proceeding of 8th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2013), 2013: 219-226.

APPENDIX B
SOURCE CODE OF DEADLOCK DETECTION TOOL

B.1. Model Package

The source code of the Deadlock Detection Algorithm is implemented as the Deadlock DetectionAlgorithm.java in the thesis.deadlockdetection.model package. It is shown in Table 14.

Table 14 The source code of the DeadlockDetectionAlgorithm.java

```
package thesis.deadlockdetection.model;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Scanner;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;

public class DeadlockDetectionAlgorithm {

	private PrintWriter pw = null;
	private PrintWriter pw2 = null;
	public File threadCountReport = new File(
			"D:\\DDReport\\ThreadCountReport.txt");
	boolean parallelCondBool = false;
	boolean escapingCondBool = false;
	boolean reachableCondBool = false;
	boolean aliasingCondBool = false;
	boolean superfluousCondBool = false;
	boolean nonguardedCondBool = false;
	boolean cyclicCondBool = false;

	File parallelConTrueFile = new
File("D:\\DDReport\\ParallelCondTrue.txt");
	File esCondTrueFile = new File("D:\\DDReport\\EsCondTrue.txt");
```

```
        File esCondTrueFileTemp = new
File("D:\\DDReport\\EsCondTrueTemp.txt");
        File reachableConTrueFile = new
File("D:\\DDReport\\ReachableConTrue.txt");
        File aliasingConTrueFile = new
File("D:\\DDReport\\AliasingCondTrue.txt");
        File superfluousCondTrueFile = new File(
                    "D:\\DDReport\\SuperflousCondTrueFile.txt");
        File nonGuardedCondTrueFile = new File(
                    "D:\\DDReport\\NonGuardedCondTrue.txt");
        File cyclicCondTrueFile = new
File("D:\\DDReport\\CyclicCondTrue.txt");

        PrintWriter pwPCondTrue = null;
        PrintWriter pwEsCondTrue = null;
        PrintWriter pwReachableCondTrue = null;
        PrintWriter pwAliasingCondTrue = null;
        PrintWriter pwSuperflousCondTrue = null;
        PrintWriter pwNonGuardedCondTrue = null;
        PrintWriter pwCyclicCondTrue = null;

        public DeadlockDetectionAlgorithm(File sourceCode, File
exportedSource) {

                System.out.println("--- DD Algo ---");

                File reportFolder = new File("D:\\DDReport\\");
                reportFolder.mkdirs();

                System.out.println("--- creating " +
reportFolder.getAbsolutePath()
                                + " directory ---");

                try {
                        pw = new PrintWriter(new
FileWriter(threadCountReport));

                        System.out.println("--- countThreads() ---");
                        countThreads(exportedSource);

                } catch (IOException e) {
                        e.printStackTrace();
                } finally {
                        if (pw != null) {
                                pw.close();
                        }
                }

                File folderForSyncInfo = new File("D:\\DDSyncInfo\\");
                folderForSyncInfo.mkdirs();
                System.out.println(folderForSyncInfo.getAbsolutePath()
                                + " directory is created.");

                try {

                        pwPCondTrue = new PrintWriter(new
FileWriter(parallelConTrueFile));
                    pwEsCondTrue = new PrintWriter(new
FileWriter(esCondTrueFile));
                    pwReachableCondTrue = new PrintWriter(new
FileWriter(
                                 reachableConTrueFile));
                    pwAliasingCondTrue = new PrintWriter(new
FileWriter(
                                 aliasingConTrueFile));
                    pwSuperflousCondTrue = new PrintWriter(new
FileWriter(
                                 superfluousCondTrueFile));
                    pwNonGuardedCondTrue = new PrintWriter(new
FileWriter(
                                 nonGuardedCondTrueFile));
                    pwCyclicCondTrue = new PrintWriter(new FileWriter(
                                 cyclicCondTrueFile));

                    System.out.println("--- collectSyncObjs() ---");
                    collectSyncObjs(exportedSource);

             } catch (Exception e) {

                    e.printStackTrace();

             } finally {

                    if (pwPCondTrue != null) {
                          pwPCondTrue.close();
                    }

                    if (pwEsCondTrue != null) {
                          pwEsCondTrue.close();
                    }

                    if (pwReachableCondTrue != null) {
                          pwReachableCondTrue.close();
                    }

                    if (pwAliasingCondTrue != null) {
                          pwAliasingCondTrue.close();
                    }

                    if (pwSuperflousCondTrue != null) {
                          pwSuperflousCondTrue.close();
                    }

                    if (pwNonGuardedCondTrue != null) {
                          pwNonGuardedCondTrue.close();
                    }

                    if (pwCyclicCondTrue != null) {
                          pwCyclicCondTrue.close();
                    }
             }
```

```
	}


	public void syncObjOrder(ArrayList<String> syncObjs) {

		Object synObOrder[] = null;
		int value = 0;
		Map<String, Integer> map = new HashMap<String,
Integer>();
		for (String s : syncObjs) {
			if (s.equals("{") | s.equals("}")) {
				continue;
			} else {
				map.put(s, value++);
			}
		}

		synObOrder = new Object[map.size()];
		synObOrder = map.keySet().toArray();
		for (int i = 0; i < synObOrder.length; i++) {
			System.out.println(synObOrder[i]);
			for (String s : syncObjs) {
				if (s.equals("{") | s.equals("}")) {
					continue;
				} else if (s.equals(synObOrder[i])) {

				}
			}
		}

	}

	public void formatEsConTrueFile(File esCondTrueFile) throws
Exception {

		SortedSet<String> sortSet = new TreeSet<String>();
		BufferedReader br = new BufferedReader(new
FileReader(esCondTrueFile));
		String line = "";

		while ((line = br.readLine()) != null) {
			sortSet.add(line);
		}

		if (br != null) {
			br.close();
		}

		PrintWriter pwEsCon = new PrintWriter(
				new FileWriter(esCondTrueFileTemp));
		Iterator<String> iter = sortSet.iterator();
		while (iter.hasNext()) {
			String type = (String) iter.next();

			pwEsCon.println(type);
		}
```

```
              if (pwEsCon != null) {
                     pwEsCon.close();
              }

              FileInputStream fis = new
FileInputStream(esCondTrueFileTemp);
              FileOutputStream fos = new
FileOutputStream(esCondTrueFile);

              byte[] buffer = new byte[4096];
              int bytesRead;

              while ((bytesRead = fis.read(buffer)) != -1) {
                     fos.write(buffer, 0, bytesRead);
              }

              if (fis != null) {
                     fis.close();
              }
              if (fos != null) {
                     fos.close();
              }
       }

       public void syncObjOrderCollecting(String file,
ArrayList<String> syncObj) {

       }

       public ArrayList<String> aliasingCond(String file,
                     ArrayList<String> syncObjs){

              Map<String, String> aliasingObjs = new HashMap<String,
String>();


                     System.out.println("= = = Aliasing Condition = =
=");


                     BufferedReader br = null;
                     String line = "";
                     String[] lineSplitString = null;

                     int newLength = 0;
                     try {
                            br = new BufferedReader(new FileReader(new
File(file)));
                            try {
                                   while ((line = br.readLine()) !=
null) {

                                          if (line.contains("=") &
!line.contains("new")) {
                                                 lineSplitString =
line.split(" ");
                                                 for (int i = 0; i <
lineSplitString.length; i++) {
                                                    if
(lineSplitString[i].contains("=")
                                                                 &
!lineSplitString[i].contains("new")) {
                                                          newLength =
i;
                                                          String[]
aliasingLineSplitString = new String[3];
      System.arraycopy(lineSplitString,
      newLength - 1,
      aliasingLineSplitString, 0, 3);
                                                          if
(aliasingLineSplitString[2]
      .contains("\"")) {
      aliasingLineSplitString[2] = aliasingLineSplitString[2]
            .replace("\"", " ").trim();
                                                          }
                                                          if
(aliasingLineSplitString[2]
      .contains(";")) {
      aliasingLineSplitString[2] = aliasingLineSplitString[2]
            .replace(";", " ").trim();
                                                          }

      aliasingObjs.put(
      aliasingLineSplitString[0],
      aliasingLineSplitString[2]);
                                                          System.out
      .println(aliasingLineSplitString[0]
                  + " Object is aliasing with "
                  + aliasingLineSplitString[2]);
                                                    }
                                              }
                                        } else if (line.contains("=")
& line.contains("new")) {
                                              lineSplitString =
line.split(" ");
                                              String[] tempArray = new
```

```
String[2];
                                                for (int i = 0; i <
lineSplitString.length; i++) {
                                                        if
(lineSplitString[i].contains("=")) {

        tempArray[0] = lineSplitString[i - 1];

        tempArray[1] = lineSplitString[lineSplitString.length - 1];
                                                        }
                                                }

                                                if
(tempArray[1].contains(";")) {
                                                        tempArray[1] =
tempArray[1].replace(";", " ")

        .trim();
                                                }

                                                if
(tempArray[1].contains(":")) {
                                                        tempArray[1] =
tempArray[1].replace(":", " ")

        .trim();
                                                }

        System.out.println(tempArray[0]
                                                                + " Object
is aliasing with "
                                                                +
tempArray[1]);

        aliasingObjs.put(tempArray[0], tempArray[1]);

                                        }

                                }

                        } catch (IOException e) {

                                e.printStackTrace();
                        }

                } catch (FileNotFoundException e) {

                        e.printStackTrace();
                } finally {
                        if (br != null) {
                                try {
                                        br.close();
                                } catch (IOException e) {
```

```
                                                e.printStackTrace();
                                        }
                                }
                        }

                System.out.println(aliasingObjs);
                Map<String, String> aliasingObjsTemp = new
HashMap<String, String>();
                System.out.println(syncObjs);
                for (String s : syncObjs) {

                        for (Map.Entry<String, String> entry :
aliasingObjs.entrySet()) {
                                if (entry.getKey().equals(s) |
entry.getValue().equals(s)) {
                                        aliasingObjsTemp.put(entry.getKey(),
entry.getValue());
                                }
                        }
                }

                System.out.println("Aliasing Obj Check: " +
aliasingObjsTemp);

                Map<String, String> aliasingObjsTemp2 = new
HashMap<String, String>();

                for (Map.Entry<String, String> entry :
aliasingObjsTemp.entrySet()) {
                        if ((!entry.getKey().contains(entry.getValue()))
                                        &
(!entry.getValue().contains("Object()"))) {
                                aliasingObjsTemp2.put(entry.getKey(),
entry.getValue());
                        }
                }

                System.out.println("Only Aliasing Objs " +
aliasingObjsTemp2);

                boolean aliasingFlag = false;
                if (!aliasingObjsTemp2.isEmpty()) {
                        System.out.println("There are Aliasing Objects "
                                        + aliasingObjsTemp2.size() + "
couple/s.");
                        aliasingFlag = true;
                } else {
                        System.out.println("No Aliasing Object.");
                        aliasingFlag = false;
                }
                ArrayList<String> syncObjsTemp = new
ArrayList<String>();
                if (aliasingFlag) {
                        for (Map.Entry<String, String> entry :
aliasingObjsTemp2.entrySet()) {
```

```
                                        for (int i = 0; i < syncObjs.size(); i++) {
                                                if
(entry.getKey().contains(syncObjs.get(i))) {
                                                        syncObjsTemp.add(i,
entry.getValue());
                                                } else {
                                                        syncObjsTemp.add(i,
syncObjs.get(i));
                                                }
                                        }
                                }

                                if (syncObjsTemp.size() > syncObjs.size()) {
                                        syncObjsTemp = new
ArrayList<String>(syncObjsTemp.subList(0,
                                                        syncObjs.size() - 1));
                                }

                                System.out.println("Replace syncObjs with Aliasing
Object = "
                                                + syncObjsTemp);

                        } else {
                                syncObjsTemp = syncObjs;
                                System.out.println("Same syncObjs = " +
syncObjsTemp);
                        }

                        aliasingCondBool = aliasingFlag;

                        if (aliasingFlag) {

                                pwAliasingCondTrue.append(file + " \n");

                        }

                        return syncObjsTemp;

                }

                public void collectSyncObjs(File exportedSource) {

                        for (File exportedSourceFile :
exportedSource.listFiles()) {

                                if (exportedSourceFile.isDirectory()) {

                                        collectSyncObjs(exportedSourceFile);

                                } else {

                                        if
(exportedSourceFile.getName().contains(".java")) {

                                                ArrayList<String> syncObjs = new
ArrayList<String>();
```

```
                                 int syncCountThis = 0;
                                 int openBlock = 0;
                                 int closeBlock = 0;

                                 PrintWriter pw = null;
                                 BufferedReader br = null;

                                 String exportedSourceFileName =
exportedSourceFile
                                              .getName();
                                 String syncInfoFileName =
"D:\\DDSyncInfo\\"
                                              +
exportedSourceFileName;
                                 File syncInfoFile = new
File(syncInfoFileName);

                                 try {

                                        pw = new
PrintWriter(syncInfoFile);

      System.out.println(syncInfoFile.getAbsolutePath());

                                        br = new BufferedReader(new
FileReader(

      exportedSourceFile));

                                        String line = "";

                                        while ((line = br.readLine())
!= null) {

                                              if
((line.contains("synchronized(")
                                                            |
line.contains("synchronized (") | line

      .contains("public synchronized "))
                                                            &
!line.contains("//")) {

                                                     if
(line.contains("synchronized(this)")
                                                                   |
line.contains("synchronized (this)")) {


      syncCountThis++;

      syncObjs.add("this");

                                                     } else if
(line.contains("synchronized(")
line.contains("synchronized (")) {
                                                          String
syncObjName = line.substring(
      line.indexOf("(") + 1,
      line.indexOf(")")).trim();

      syncObjs.add(syncObjName);
                                                    } else if
(line.contains("synchronized")
                                                                &
!line.contains("synchronized (")
                                                                &
!line.contains("synchronized(")) {
                                                          String
syncObjName = line.substring(
      line.lastIndexOf("(") + 1,
      line.lastIndexOf(")")).trim();
                                                          syncObjName
= syncObjName.substring(
      syncObjName.lastIndexOf(" "),
      syncObjName.length()).trim();

      syncObjs.add(syncObjName);
                                                    }
                                              } else if
(line.contains("{")) {
                                                    openBlock++;

      syncObjs.add("{");
                                              } else if
(line.contains("}")) {
                                                    closeBlock++;

      syncObjs.add("}");
                                              }
                                        }
                                        pw.println(syncObjs);
                                        System.out.println("----------
original-------------");
                                        System.out.println(syncObjs);
                                        System.out.println("----------
original-------------");

                                        ArrayList<String>
revisedSyncObjs = aliasingCond(

      exportedSourceFile.getAbsolutePath(), syncObjs);

                                        parallelCondBool =
parallelCond(

      syncInfoFile.getAbsolutePath(), revisedSyncObjs);

                                        escapingCondBool =
escapingCond(

      syncInfoFile.getAbsolutePath(), revisedSyncObjs);
                                        try {


                                              reachableCondBool =
reachableCond(

      syncInfoFile.getAbsolutePath(), revisedSyncObjs);

                                        } catch (Exception e) {
                                              e.printStackTrace();
                                        }

                                        superfluousCondBool =
superfluousCond(

      exportedSourceFile.getAbsolutePath(),
                                                    revisedSyncObjs);

                                        nonguardedCondBool =
nonGuardedCond(

      exportedSourceFile.getAbsolutePath(),
                                                    revisedSyncObjs);

                                        cyclicCondBool = cyclicCond(

      exportedSourceFile.getAbsolutePath(),
                                                    revisedSyncObjs);


                                        System.out.println();

                                  } catch (IOException e) {

                                        e.printStackTrace();
```

```
                    } finally {
                        if (pw != null) {
                            pw.close();
                        }
                    }

                }
            }
        }
    }


    public boolean superfluousCond(String absolutePath,
            ArrayList<String> revisedSyncObjs)
            {

        System.out.println("===== Superfluous Condition =====");

        boolean superfluousBool = false;

            ArrayList<Integer> index = new
ArrayList<Integer>();
            ArrayList<Boolean> doubleLockList = new
ArrayList<Boolean>();

            for (int i = 0; i < revisedSyncObjs.size(); i++) {
                if (i == (revisedSyncObjs.size() - 1)) {
                    break;
                } else {
                    if
(revisedSyncObjs.get(i).equals("}")
                            ||
revisedSyncObjs.get(i).equals("{")
                            || revisedSyncObjs.get(i
+ 1).equals("}")
                            || revisedSyncObjs.get(i
+ 1).equals("{")) {
                        continue;
                    } else {
                        System.out.println(i);
                        index.add(i);

    doubleLockList.add(arrayListToken(

    revisedSyncObjs.get(i),

    revisedSyncObjs.get(i + 1)));

                    }
                }
            }

            System.out.println("Double lock list = " +
doubleLockList);

            int countTrue = 0;
```

```
                    for (boolean boolEle : doubleLockList) {
                          if (boolEle == true) {
                                countTrue++;
                          }
                    }

                    if (countTrue > 0) {
                          System.out.println("Deadlock.");
                          superfluousBool = true;
                    } else {
                          System.out.println("No Deadlock");
                          superfluousBool = false;

                          pwSuperflousCondTrue.append(absolutePath +
"\n");

                    }

       System.out.println("======================================");

             return superfluousBool;
       }

       public static boolean arrayListToken(String a, String b) {
             boolean doubleLock = true;
             System.out.println("Check group = " + a + ", " + b);
             if (a.equals(b)) {
                    doubleLock = true;
                    System.out
                          .println("No deadlock/Superfluous
Condition - Double Lock.");
             } else {
                    doubleLock = false;
                    System.out
                          .println("Potential deadlock/No
Superfluous Condition - No Double Lock.");
             }

             return doubleLock;
       }

       public boolean cyclicCond(String absolutePath,
                    ArrayList<String> revisedSyncObjs)
       {

             boolean cyclicCondResult = false;
             ArrayList<Integer> index = new ArrayList<Integer>();

             System.out.println("===========Reverse
Order=============");

                    for (int i = 0; i < revisedSyncObjs.size(); i++) {
                          if (revisedSyncObjs.get(i).equals("this"))
{
                                revisedSyncObjs.set(i, "}");
                          }
```

```
            }

            for (int i = 0; i < revisedSyncObjs.size(); i++) {
                if (!(revisedSyncObjs.get(i).equals("{") ||
revisedSyncObjs.get(
                        i).equals("}"))) {
                    index.add(i);
                }
            }

            System.out.println(index);

            BufferedReader br = null;
            String line = "";
            int numOfThread = 0;
            String fileSearch = absolutePath.substring(
                    absolutePath.lastIndexOf("\\") + 1,
absolutePath.length())
                    .trim();
            Scanner scanner = null;
            try {

                br = new BufferedReader(new
FileReader(threadCountReport));
                scanner = new Scanner(br);

                while (scanner.hasNext()) {
                    if
(scanner.next().equals(fileSearch)) {
                        scanner.next();
                        if (scanner.hasNextInt()) {
                            numOfThread =
scanner.nextInt();

    System.out.println("There are " + numOfThread + " threds.");
                            break;
                        }
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (scanner != null) {
                    scanner.close();
                }
                try {
                    if (br != null) {
                        br.close();
                    }

                } catch (IOException e) {

                    e.printStackTrace();
                }
            }
```

```
                    int groupMaxSize = 0;

                    for (int i = 0; i < index.size(); i++) {

                          groupMaxSize +=
checkGroupSize(index.get(i), index.get(i++));
                    }

                    System.out.println("GroupObjSize = " +
groupMaxSize);

                    if (numOfThread == 2) {

                          if (groupMaxSize == 1) {

                                 cyclicCondResult = false;
                                 System.out.println("No Deadlock");

                          } else if (groupMaxSize == 3 & index.size()
== 5) {

                                 if (index.get(groupMaxSize - 1) + 1 <
index
                                              .get(groupMaxSize)) {

                                        if
(revisedSyncObjs.get(index.get(0)).equals(

      revisedSyncObjs.get(index.get(groupMaxSize)))) {

                                              cyclicCondResult =
false;
                                              System.out.println("No
Deadlock");
                                        } else {

                                              if
(revisedSyncObjs.get(index.get(0))

      .equals(revisedSyncObjs.get(index.get(index

      .size() - 1)))) {

                                                    if
(revisedSyncObjs.get(

      index.get(index.size() - 2)).equals(

      revisedSyncObjs.get(index.get(1)))
                                                                  |
revisedSyncObjs.get(

             index.get(index.size() - 2))

             .equals(revisedSyncObjs
                              .get(index.get(2)))) {


       cyclicCondResult = true;

       System.out.println("Deadlock");

                                                          } else {


       cyclicCondResult = false;

       System.out.println("No Deadlock");
                                                          }
                                                   } else if
(revisedSyncObjs.get(index.get(1))

       .equals(revisedSyncObjs.get(index.get(index

       .size() - 1)))
                                                                 &
revisedSyncObjs.get(

       index.get(groupMaxSize - 1))

       .equals(revisedSyncObjs.get(index

                    .get(groupMaxSize)))) {

                                                          cyclicCondResult
= true;

       System.out.println("Deadlock");
                                                   } else {

                                                          cyclicCondResult
= false;

       System.out.println("No Deadlock");
                                                   }
                                             }
                                       } else {

                                             int firstGroupSize =
groupMaxSize - 1;


                                             if
(revisedSyncObjs.get(index.get(0)).equals(

       revisedSyncObjs.get(index.get(firstGroupSize)))) {

                                                   cyclicCondResult =
false;
                                                   System.out.println("No
Deadlock");
```

```
                                        } else if
(!revisedSyncObjs.get(index.get(0)).equals(

      revisedSyncObjs.get(index.get(firstGroupSize)))) {
                                              if
((revisedSyncObjs.get(index.get(0))

      .equals(revisedSyncObjs.get(index.get(index

      .size() - 2))) | revisedSyncObjs

      .get(index.get(0)).equals(

      revisedSyncObjs.get(index.get(index

                   .size() - 1))))
                                                           &
revisedSyncObjs.get(

      index.get(firstGroupSize - 1))

      .equals(revisedSyncObjs.get(index

                   .get(firstGroupSize)))) {

                                                    cyclicCondResult
= true;

      System.out.println("Deadlock");

                                              } else if
(revisedSyncObjs.get(index.get(0))

      .equals(revisedSyncObjs.get(index.get(index

      .size() - 1)))
                                                           &
(revisedSyncObjs.get(

      index.get(firstGroupSize - 1))

      .equals(revisedSyncObjs.get(index

                   .get(firstGroupSize))) | revisedSyncObjs

      .get(index.get(firstGroupSize - 1))

      .equals(revisedSyncObjs.get(index

                   .get(index.size() - 2))))) {

                                                    cyclicCondResult
= true;

      System.out.println("Deadlock");
```

```
                                              } else {

                                                    cyclicCondResult
= false;

      System.out.println("No Deadlock");
                                              }
                                       }

                                 }
                          } else if (groupMaxSize == 2 & index.size()
== 4) {

                                 if
(revisedSyncObjs.get(index.get(0)).equals(

      revisedSyncObjs.get(index.get(groupMaxSize)))) {

                                       cyclicCondResult = false;
                                       System.out.println("No
Deadlock");

                                 } else {
                                       if
(revisedSyncObjs.get(index.get(0))

      .equals(revisedSyncObjs.get(index.get(index

      .size() - 1)))
                                                    &
revisedSyncObjs.get(index.get(1)).equals(

      revisedSyncObjs.get(index

             .get(groupMaxSize)))) {

                                              cyclicCondResult = true;

      System.out.println("Deadlock");

                                       } else {

                                              cyclicCondResult =
false;
                                              System.out.println("No
Deadlock");
                                       }
                                 }
                          } else if (groupMaxSize == 3 & index.size()
== 6) {
                                 if
(revisedSyncObjs.get(index.get(0)).equals(

      revisedSyncObjs.get(index.get(groupMaxSize)))) {
```

```
                                                cyclicCondResult = false;
                                                System.out.println("No
Deadlock");
                                        } else if
(!revisedSyncObjs.get(index.get(0)).equals(

        revisedSyncObjs.get(index.get(groupMaxSize)))) {

                                                if
(revisedSyncObjs.get(index.get(0))

        .equals(revisedSyncObjs.get(index.get(index

        .size() - 1)))
                                                                &
(revisedSyncObjs.get(index.get(1)).equals(

        revisedSyncObjs.get(index

                .get(groupMaxSize))) | revisedSyncObjs

        .get(index.get(1))

        .equals(revisedSyncObjs.get(index

                .get(groupMaxSize + 1))))
                                                                &
(revisedSyncObjs.get(index.get(2)).equals(

        revisedSyncObjs.get(index

                .get(groupMaxSize))) | revisedSyncObjs

        .get(index.get(2))

        .equals(revisedSyncObjs.get(index

                .get(groupMaxSize + 1))))) {

                                                        cyclicCondResult = true;

        System.out.println("Deadlock");

                                                } else if
(revisedSyncObjs.get(index.get(0))

        .equals(revisedSyncObjs.get(index

        .get(groupMaxSize + 1)))
                                                                &
(revisedSyncObjs.get(index.get(1)).equals(

        revisedSyncObjs.get(index

                .get(groupMaxSize))) | revisedSyncObjs
```

```
       .get(index.get(1))

       .equals(revisedSyncObjs.get(index

              .get(index.size() - 1))))
                                                            &
(revisedSyncObjs.get(index.get(2)).equals(

       revisedSyncObjs.get(index

              .get(groupMaxSize))) | revisedSyncObjs

       .get(index.get(2))

       .equals(revisedSyncObjs.get(index

              .get(index.size() - 1))))) {

                                                 cyclicCondResult = true;

       System.out.println("Deadlock");

                                          } else {

                                                 cyclicCondResult =
false;
                                                 System.out.println("No
Deadlock");
                                          }

                                   }

                            }

                     } else if (numOfThread == 3) {
                            System.out.println("else if 3 threads.");
                            System.out.println("lock size = " +
groupMaxSize);
                            if (groupMaxSize == 3 & index.size() == 6)
{

                                   if (index.get(groupMaxSize) <
(index.get(groupMaxSize) + 1)) {
                                          System.out.println("2 locks, 3
threads ");

                                          int currentGroupSize =
groupMaxSize - 1;

                                          if
(revisedSyncObjs.get(index.get(0)).equals(

       revisedSyncObjs.get(index.get(2)).equals(

       revisedSyncObjs.get(index.get(4))))) {
```

```
                                              cyclicCondResult =
false;
                                              System.out.println("No
Deadlock");

                                       } else {

                                              if
(revisedSyncObjs.get(index.get(0))

      .equals(revisedSyncObjs.get(index.get(index

      .size() - 1)))
                                                            &
revisedSyncObjs.get(index.get(1)).equals(

      revisedSyncObjs.get(index.get(2)))
                                                            &
revisedSyncObjs.get(index.get(3)).equals(

      revisedSyncObjs.get(index.get(4)))) {

                                                     cyclicCondResult
= true;

      System.out.println("Deadlock");

                                              } else if
(revisedSyncObjs.get(index.get(0))

      .equals(revisedSyncObjs.get(index.get(3)))
                                                            &
revisedSyncObjs.get(index.get(1)).equals(

      revisedSyncObjs.get(index.get(4)))
                                                            &
revisedSyncObjs.get(index.get(2)).equals(

      revisedSyncObjs.get(index.get(5)))) {

                                                     cyclicCondResult
= true;

      System.out.println("Deadlock");

                                              } else if
(revisedSyncObjs.get(index.get(2))

      .equals(revisedSyncObjs.get(index.get(5)))
                                                            &
revisedSyncObjs.get(index.get(3)).equals(

      revisedSyncObjs.get(index.get(4)))) {

                                                     cyclicCondResult
= true;
```

```
      System.out.println("Deadlock");
                                            } else if
(revisedSyncObjs.get(index.get(0))
      .equals(revisedSyncObjs.get(index.get(5)))
                                                       &
revisedSyncObjs.get(index.get(1)).equals(
      revisedSyncObjs.get(index.get(4)))) {
                                                  cyclicCondResult
= true;
      System.out.println("Deadlock");
                                            } else if
(revisedSyncObjs.get(index.get(0))
      .equals(revisedSyncObjs.get(index.get(3)))
                                                       &
revisedSyncObjs.get(index.get(1)).equals(
      revisedSyncObjs.get(index.get(2)))) {
                                                  cyclicCondResult
= true;
      System.out.println("Deadlock");
                                            } else {
                                                  cyclicCondResult
= false;
      System.out.println("No deadlock.");
                                            }
                                      }
                                }
                          } else if (index.size() == 5) {
                                if
(revisedSyncObjs.get(index.get(0)).equals(
      revisedSyncObjs.get(index.get(index.size() - 1)))) {
                                      if
(revisedSyncObjs.get(index.get(1))
      .equals(revisedSyncObjs.get(index.get(index
      .size() - 2)))) {
                                            cyclicCondResult = true;
      System.out.println("Deadlock");

                                      } else {

                                            cyclicCondResult =
false;
                                            System.out.println("No
Deadlock");
                                      }
                                } else {

                                      cyclicCondResult = false;
                                      System.out.println("No
Deadlock");
                                }
                          }

                    } else if (numOfThread > 3) {
                          System.out.println(index);
                          if
(revisedSyncObjs.get(index.get(0)).equals(

      revisedSyncObjs.get(index.get(index.size() - 1)))) {

                                if
(revisedSyncObjs.get(index.get(1)).equals(

      revisedSyncObjs.get(index.get(groupMaxSize)))) {
                                      cyclicCondResult = true;

      System.out.println("Deadlock");

                                } else {

                                      cyclicCondResult = false;
                                      System.out.println("No
Deadlock");
                                }

                          } else {

                                cyclicCondResult = false;
                                System.out.println("No Deadlock");
                          }

                    }

                 System.out

      .println("=======================================================
=");

                 if (cyclicCondResult) {
                       pwCyclicCondTrue.append(absolutePath + " \n");
                 }

                 return cyclicCondResult;
      }

      public int checkGroupSize(int x, int y) {
                 int maxCount = 0;

                 if ((x++) == y) {
                       maxCount++;
                 }

                 return maxCount;
      }

      public boolean nonGuardedCond(String absolutePath,
                       ArrayList<String> revisedSyncObjs) {

                 System.out.println("=====Non-Guarded Condition======");
                 boolean nonguarded = false;
                       ArrayList<Integer> index = new
ArrayList<Integer>();

                       for (int i = 0; i < revisedSyncObjs.size(); i++) {
                          if (!(revisedSyncObjs.get(i).equals("{") |
revisedSyncObjs.get(
                                      i).equals("}"))) {
                                index.add(i);
                          }
                       }

                       BufferedReader br = null;
                       String line = "";
                       int numOfThread = 0;
                       String fileSearch = absolutePath.substring(
                                absolutePath.lastIndexOf("\\") + 1,
absolutePath.length())
                                .trim();
                       Scanner scanner = null;
                       try {

                          br = new BufferedReader(new
FileReader(threadCountReport));
                          scanner = new Scanner(br);

                          while (scanner.hasNext()) {
                                if
(scanner.next().equals(fileSearch)) {
                                      scanner.next();
                                      if (scanner.hasNextInt()) {
                                            numOfThread =
scanner.nextInt();

      System.out.println("There are " + numOfThread
                                                      + "
threds.");
```

```
                                                        break;
                                                }
                                        }
                                }
                        } catch (Exception e) {
                                e.printStackTrace();
                        } finally {
                                if (scanner != null) {
                                        scanner.close();
                                }
                                try {
                                        if (br != null) {
                                                br.close();
                                        }

                                } catch (IOException e) {

                                        e.printStackTrace();
                                }
                        }

                        int groupMaxSize = 0;

                        for (int i = 0; i < index.size(); i++) {

                                groupMaxSize +=
checkGroupSize(index.get(i), index.get(i++));
                        }

                        System.out.println("GroupObjSize = " +
groupMaxSize);

                        if (numOfThread == 2) {

                                if (groupMaxSize == 2) {
                                        if
(revisedSyncObjs.get(index.get(0)).equals(

        revisedSyncObjs.get(index.get(groupMaxSize)))) {
                                                System.out.println("No
Deadlock/guarded lock");
                                                nonguarded = false;
                                        } else {

        System.out.println("Deadlock");
                                                nonguarded = true;
                                        }

                                } else if (groupMaxSize == 3) {

                                        if (index.size() == 5) {

                                                if (index.get(groupMaxSize -
1) + 1 == index

        .get(groupMaxSize)) {
```

```
                                              groupMaxSize =
groupMaxSize - 1;

                                              if
(revisedSyncObjs.get(index.get(0))

      .equals(revisedSyncObjs.get(index

      .get(groupMaxSize)))) {

      System.out.println("No Deadlock/guarded lock");
                                                    nonguarded =
false;
                                              } else {

      System.out.println("Deadlock");
                                                    nonguarded =
true;
                                              }
                                        } else {

                                              if
(revisedSyncObjs.get(index.get(0))

      .equals(revisedSyncObjs.get(index

      .get(groupMaxSize)))) {

      System.out.println("No Deadlock/guarded lock");
                                                    nonguarded =
false;
                                              } else {

      System.out.println("Deadlock");
                                                    nonguarded =
true;
                                              }
                                        }

                                  } else if (index.size() == 6) {

                                        if
(revisedSyncObjs.get(index.get(0)).equals(

      revisedSyncObjs.get(index.get(groupMaxSize)))) {
                                              System.out.println("No
Deadlock/guarded lock");
                                              nonguarded = false;
                                        } else {

      System.out.println("Deadlock");
                                              nonguarded = true;
                                        }
                                  }
                            }
```

```
            } else if (numOfThread == 3) {

                groupMaxSize = groupMaxSize - 1;

                System.out.println(groupMaxSize);

                if
(revisedSyncObjs.get(index.get(0)).equals(

    revisedSyncObjs.get(index.get(groupMaxSize)))) {

                    if
(revisedSyncObjs.get(index.get(0)).equals(

    revisedSyncObjs.get(index.get(groupMaxSize + 2)))) {
                        System.out.println("No
Deadlock/guarded lock");

                        nonguarded = false;

                    } else {


    System.out.println("Deadlock");
                        nonguarded = true;
                    }

                } else {

                    System.out.println("Deadlock");
                    nonguarded = true;

                }
            } else {

                System.out.println("Deadlock");
                nonguarded = true;
            }

        if (nonguarded) {

            pwNonGuardedCondTrue.append(absolutePath + " \n");
        }


    System.out.println("=========================================")
;
        return nonguarded;
    }

    public static boolean checkEqual(ArrayList<String> arrayList,
int a, int b) {

        boolean guardedLockBool = false;

        if (arrayList.get(a).equals(arrayList.get(b))) {
            guardedLockBool = true;
            System.out.println("Guarded Lock");
        } else {
            guardedLockBool = false;
            System.out.println("Non-Guarded Lock");
        }

        return guardedLockBool;
    }

    public void syncObjGroup(String file, ArrayList<String>
syncObjs) {

        String[] syncGroupArray[] = new String[2][3];

        Iterator<String> iter = syncObjs.iterator();

        for (int i = 0; i < syncGroupArray.length; i++) {
            for (int j = 0; j < syncGroupArray[i].length; j++)
{
                while (iter.hasNext()) {
                    if (!iter.next().equals("}") &
!iter.next().equals("{")) {
                        syncGroupArray[i][j] =
iter.next();
                    }
                }
            }
        }

    }

    public boolean reachableCond(String file, ArrayList<String>
syncObj
                    )throws Exception {

            System.out.println("--- Reachable Condition ---");

            String fileSearch =
file.substring(file.lastIndexOf("\\") + 1,
                    file.length()).trim();

            int numOfThread = 0;

            BufferedReader br = new BufferedReader(new
FileReader(
                    threadCountReport));
            Scanner scanner = new Scanner(br);

            while (scanner.hasNext()) {
                if (scanner.next().equals(fileSearch)) {
                    scanner.next();
                    if (scanner.hasNextInt()) {
                        numOfThread =
scanner.nextInt();
                        break;
```

```
                                    }
                              }
                        }

                        Map<String, Integer> map = new HashMap<String,
Integer>();

                        for (String entry : syncObj) {
                              Integer count = map.get(entry);
                              map.put(entry, (count == null) ? 1 : count
+ 1);
                        }

                        for (Map.Entry<String, Integer> entry :
map.entrySet()) {

                              if (!(entry.getKey().equals("{") |
entry.getKey().equals("}")))

                                    if (numOfThread > 0 &
entry.getValue() > 0) {


      pwReachableCondTrue.append(fileSearch + " \n");

                                          System.out.println("Reachable
Cond - - - " + fileSearch
                                                      + " Deadlock ---
");

                                          reachableCondBool = true;
                                          break;

                                    } else {

                                          System.out.println("Reachable
Cond - - - " + file
                                                      + " No Deadlock -
- -");
                                          reachableCondBool = false;

                                    }
                        }

                        if (br != null) {
                              br.close();
                        }
                        scanner.close();

                  return reachableCondBool;
      }

      public boolean escapingCond(String file, ArrayList<String>
syncObj)
                                    throws FileNotFoundException {
```

```
                    System.out.println("--- Escaping Condition ---");

                    Map<String, Integer> map = new HashMap<String,
Integer>();

                    for (String entry : syncObj) {
                          Integer count = map.get(entry);
                          map.put(entry, (count == null) ? 1 : count
+ 1);
                    }

                    for (Map.Entry<String, Integer> entry :
map.entrySet()) {

                          if (!(entry.getKey().equals("{") |
entry.getKey().equals("}")))
                                if (entry.getValue() > 0) {
                                      System.out.println("key : " +
entry.getKey()
                                                  + " value : " +
entry.getValue());
                                      pwEsCondTrue.append(file + "
\n");

                                      System.out.println("Escaping
Cond - - - " + file
                                                  + " Deadlock ---
");

                                      escapingCondBool = true;
                                      break;

                                } else {
                                      escapingCondBool = false;
                                      System.out.println("Escaping
Cond - - -" + file
                                                  + " No Deadlock -
--");
                                }
                          break;
                    }

             return escapingCondBool;

       }

       public boolean parallelCond(String file, ArrayList<String>
syncObj)
                    throws IOException {

             System.out.println("--- Parallel Condition ---");
             int groupOfLocks = 0;
             int openBlock = 0;
             int closeBlock = 0;
             int lockedObj = 0;
```

```
              BufferedReader br = null;
              String line = "";
              int numOfThread = 0;
              String fileSearch =
file.substring(file.lastIndexOf("\\") + 1,
                          file.length()).trim();
              Scanner scanner = null;
              try {

                     br = new BufferedReader(new
FileReader(threadCountReport));
                     scanner = new Scanner(br);

                     while (scanner.hasNext()) {
                            if (scanner.next().equals(fileSearch)) {
                                   scanner.next();
                                   if (scanner.hasNextInt()) {
                                          numOfThread =
scanner.nextInt();
                                          System.out.println("There are
" + numOfThread

                                                        + " threds.");
                                          break;
                                   }
                            }
                     }

                     for (String entry : syncObj) {
                            if (entry.equals("{")) {
                                   openBlock++;
                            } else if (entry.equals("}")) {
                                   closeBlock++;
                            } else {
                                   lockedObj++;
                            }
                     }
                     System.out.println("locked Objs = " + lockedObj);

                     if ((numOfThread > 1) & (numOfThread != 0)
                                   & (lockedObj >= numOfThread)) {
                            pwPCondTrue.append(fileSearch + " \n");
                            parallelCondBool = true;
                            System.out.println("Parallel Cond - - - " +
fileSearch
                                          + " Deadlock ---");

                     } else {
                            parallelCondBool = false;
                            System.out.println("Parallel Cond - - - " +
fileSearch
                                          + " No Deadlock ---");

                     }

              } catch (FileNotFoundException e) {
```

```
            e.printStackTrace();
        } finally {
            if (br != null) {
                br.close();
            }
            scanner.close();

        }

        return parallelCondBool;

    }

    public void getFile(File exportedSource) {

        File[] fileList = exportedSource.listFiles();

        for (int i = 0; i < fileList.length; i++) {

            if (fileList[i].isDirectory()) {

                getFile(fileList[i]);

            } else {
                System.out.println(fileList[i].getName());

            }
        }

    }

    public void countThreads(File exportedSource) {

        for (File exportedFile : exportedSource.listFiles()) {

            if (exportedFile.isDirectory()) {
                countThreads(exportedFile);

            } else {

                String exportedFileName =
exportedFile.getAbsolutePath();

                String exportedClassName =
exportedFile.getName();

                if (exportedClassName.contains(".java")) {

                    File threadCountFile = new
File(exportedFileName);

                    int threadCount = 0;

                    BufferedReader br = null;
```

```
                                try {
                                        br = new BufferedReader(new
FileReader(threadCountFile));
                                        String line = null;
                                        try {
                                                while ((line =
br.readLine()) != null) {
                                                        if
(line.contains(".start()")) {

        threadCount++;
                                                        }
                                                }
                                        } catch (IOException e) {
                                                e.printStackTrace();
                                        }
                                } catch (FileNotFoundException e) {
                                        e.printStackTrace();
                                } finally {
                                        if (br != null) {
                                                try {
                                                        br.close();
                                                } catch (IOException e)
{

        e.printStackTrace();
                                                }
                                        }
                                }
                                pw.append(exportedClassName + " has "
+ threadCount
                                                + " threads are
created.\n");

                                System.out.println(exportedClassName
+ " has "
                                                + threadCount + "
threads are created.\n");
                        }
```

```
                    }

              }
        }
}
```

B.2. View Package

The source code of the View package of the Deadlock Detection Tool is named that the thesis.deadlockdetection.view package. There are 2 files that are the DeadlockDetectionTool.java and DeadlockOutputDisplay.java.

B.2.1. DeadlockDetectionTool.java

The source code of the Deadlock Detection Tool that is implemented as the DeadlockDetectionTool.java file. It is the GUI for users to interact with the Deadlock Detection Tool. The source code is shown in Table 15.

Table 15 The source code of the DeadlockDetectionTool.java

```
package thesis.deadlockdetection.view;

import java.awt.Dimension;
import java.awt.EventQueue;
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import java.awt.GridLayout;
import javax.swing.DefaultListModel;
import javax.swing.GroupLayout;
import javax.swing.GroupLayout.Alignment;
import javax.swing.JFileChooser;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.JProgressBar;
import javax.swing.ScrollPaneConstants;
import javax.swing.LayoutStyle.ComponentPlacement;
import javax.swing.SwingWorker;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.MouseListener;
import java.awt.event.WindowEvent;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Random;
import javax.swing.JSplitPane;
import javax.swing.JTextPane;
import javax.swing.JList;
import javax.swing.UIManager;
import javax.swing.border.LineBorder;
import javax.swing.text.DefaultCaret;
import java.awt.Color;
import javax.swing.JSeparator;
import thesis.deadlockdetection.model.DeadlockDetectionAlgorithm;
import thesis.deadlockdetection.controller.DataPreparation;
import javax.swing.JScrollBar;

public class DeadlockDetectionTool {

	private JFrame frmDeadlock;
	private JTextField textField;
	private JTextField textField_1;

	private JFrame pBarFrame = new JFrame();
	private Task task;
	private JProgressBar pBar = new JProgressBar(0, 100);
	private JTextField textField_2;
	private JButton btnRun = new JButton("Run");

	DeadlockOutputDisplay display = null;
	DefaultListModel<String>        dListModel         =         new
DefaultListModel<String>();
	private JList<String> list = new JList<String>(dListModel);

	private JTextArea textArea = new JTextArea();

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					DeadlockDetectionTool   window   =   new
DeadlockDetectionTool();
					window.frmDeadlock.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}
	/**
	 * Create the application.
	 */
	public DeadlockDetectionTool() {
		initialize();
	}

	/**
	 * Initialize the contents of the frame.
	 */
	private void initialize() {
		frmDeadlock = new JFrame();
		frmDeadlock.setTitle("Deadlock Detection Tool");
		frmDeadlock.setBounds(10, 10, 818, 700);
		frmDeadlock.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		JPanel panel = new JPanel();
		frmDeadlock.getContentPane().add(panel,
BorderLayout.CENTER);

		JLabel lblLocation = new JLabel(
				"Browse Location of Multithreading API : ");

		textField = new JTextField();
		textField.setColumns(10);

		textField_1 = new JTextField();

		textField_1.setColumns(10);
		textField_1.setText("");

		JButton btnBroswe = new JButton("Browse...");
		btnBroswe.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				JFileChooser fileChooser = new JFileChooser();
	fileChooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);

				fileChooser.setCurrentDirectory(new
File(textField_1.getText()));
				int			returnVal		=
fileChooser.showDialog(frmDeadlock, "Ok");
				if  (returnVal  ==  JFileChooser.APPROVE_OPTION)
{

	textField_1.setText(fileChooser.getSelectedFile()
							.getAbsolutePath());
				}
			}
		});

		btnRun.addMouseListener(new MouseAdapter() {
			@Override
			public void mouseClicked(MouseEvent e) {

				pBarFrame.setLocation(400, 400);
```

```
                            pBarFrame.setTitle("Task Progress ...");

                            pBar.setValue(0);
                            pBar.setStringPainted(true);

                            JPanel pPanel = new JPanel();
                            pPanel.setPreferredSize(new     Dimension(250,
50));

                            pPanel.add(pBar);

                            pBarFrame.getContentPane().add(pPanel);

                            pBarFrame.pack();
                            pBarFrame.setVisible(true);
                    }
             });

             btnRun.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent arg0) {
                            task = new Task();
                            task.addPropertyChangeListener(new
PropertyChangeListener() {

                                   @Override
                                   public                          void
propertyChange(PropertyChangeEvent evt) {

                                          if          ("progress"          ==
evt.getPropertyName()) {
                                                 int  progress  =  (Integer)
evt.getNewValue();
                                                 pBar.setValue(progress);
                                          }
                                   }
                            });
                            task.execute();
                    }
             });

             JPanel panel_1 = new JPanel();
             panel_1.setBorder(new LineBorder(UIManager

       .getColor("CheckBoxMenuItem.selectionBackground")));

             JLabel lblDeadlockResult = new JLabel("Deadlock Result:");
             GroupLayout gl_panel = new GroupLayout(panel);
             gl_panel.setHorizontalGroup(gl_panel
                            .createParallelGroup(Alignment.LEADING)
                            .addGroup(
                                          gl_panel.createSequentialGroup()
                                                        .addGap(23)
                                                        .addGroup(
                                          gl_panel.createParallelGroup(
                                                 Alignment.LEADING)
                                                        .addComponent(
                                                               panel_1,
         GroupLayout.PREFERRED_SIZE,760,
         GroupLayout.PREFERRED_SIZE)
                            .addGroup(
                            gl_panel.createSequentialGroup()
                                                .addGroup(
         gl_panel.createParallelGroup(
             Alignment.TRAILING)
                    .addGroup(
         gl_panel.createSequentialGroup()
             .addComponent(
                            lblDeadlockResult,
                            GroupLayout.PREFERRED_SIZE,
                            152,
                            GroupLayout.PREFERRED_SIZE)
             .addGap(359)
             .addComponent(
                            btnRun,
                            GroupLayout.PREFERRED_SIZE,
                            73,
                            GroupLayout.PREFERRED_SIZE))
                                                               .addGroup(
         gl_panel.createParallelGroup(
             Alignment.LEADING)
             .addComponent(
                            lblLocation,
                            GroupLayout.PREFERRED_SIZE,
                            364,
                            GroupLayout.PREFERRED_SIZE)
             .addComponent(
                            textField_1,
                            GroupLayout.PREFERRED_SIZE,
                            587,
                            GroupLayout.PREFERRED_SIZE)))
                                   .addPreferredGap(
         ComponentPlacement.UNRELATED)
                                   .addComponent(
                                                 btnBroswe)))
       .addContainerGap(19, Short.MAX_VALUE)));
             gl_panel.setVerticalGroup(gl_panel
                            .createParallelGroup(Alignment.LEADING)
                            .addGroup(
                                          gl_panel.createSequentialGroup()
                                                        .addContainerGap()
       .addComponent(lblLocation)
       .addPreferredGap(ComponentPlacement.UNRELATED)
                                                        .addGroup(
       gl_panel.createParallelGroup(
       Alignment.BASELINE)
       .addComponent(
                    textField_1,
                    GroupLayout.PREFERRED_SIZE,
                    GroupLayout.DEFAULT_SIZE,
                    GroupLayout.PREFERRED_SIZE)
       .addComponent(btnBroswe))
       .addPreferredGap(ComponentPlacement.RELATED)
```

```
              .addGroup(

       gl_panel.createParallelGroup(
              Alignment.TRAILING)
              .addComponent(btnRun)
              .addComponent(lblDeadlockResult))
                     .addGap(18)
                            .addComponent(panel_1,
                                   GroupLayout.DEFAULT_SIZE, 336,
                                   Short.MAX_VALUE).addContainerGap()));

              DefaultCaret caret = (DefaultCaret) textArea.getCaret();
              caret.setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);
              panel.setLayout(gl_panel);
              list.addMouseListener(new MouseAdapter() {
                     @Override
                     public  void  mouseClicked(MouseEvent  listMouseEvent)
{
                            textArea.setText("");
                            JList        theList         =         (JList)
listMouseEvent.getSource();
                            int                index                     =
theList.locationToIndex(listMouseEvent.getPoint());
                            if (index >= 0) {
                                   Object              obj               =
theList.getModel().getElementAt(index);

                                   File  file  =  new  File("D:\\DDDisplay\\"
+ obj.toString());
                                   BufferedReader br = null;
                                   String inputLine = null;

                                   try {

                                          br    =    new    BufferedReader(new
FileReader(file));
                                          try {
                                                 while     ((inputLine     =
br.readLine()) != null) {

       textArea.append(inputLine + "\n");
       textArea.setCaretPosition(textArea
                            .getDocument().getLength());
                            }
                                          } catch (IOException e) {
                                                 e.printStackTrace();
                                          }

                                   } catch (FileNotFoundException e) {
                                          e.printStackTrace();
                                   } finally {
                                          if (br != null) {
                                                 try {
                                                        br.close();
                                                 } catch (IOException e) {
                                                 e.printStackTrace();
```

```
                                                            }
                                                    }
                                            }
                                    }
                            }
                    });

                    list.setBorder(new LineBorder(UIManager
            .getColor("CheckBoxMenuItem.selectionBackground")));
                    JLabel lblNumberOfDeadlocks = new JLabel(
                                    "Number Of Detected Deadlocks:");
                    textField_2 = new JTextField();
                    textField_2.setColumns(10);
                    JLabel lblListOfFiles = new JLabel("List of files:");
                    JLabel lblSites = new JLabel("files");
                    JLabel lblDetailOfDeadlock = new JLabel(
                                    "Detail of Deadlock from selected file:");
                    GroupLayout gl_panel_1 = new GroupLayout(panel_1);
                    gl_panel_1
                                    .setHorizontalGroup(gl_panel_1
                            .createParallelGroup(Alignment.TRAILING)
                                                    .addGroup(
                                                            gl_panel_1
                                                    .createSequentialGroup()
                                                                    .addGap(25)
                                                                    .addGroup(
                                                                    gl_panel_1
                                                    .createParallelGroup(
                                            Alignment.LEADING,
                                            false)
                                                            .addComponent(
                                                    lblListOfFiles,
                                            GroupLayout.PREFERRED_SIZE,
                                            149,
                                            GroupLayout.PREFERRED_SIZE).addGroup(
                                                    gl_panel_1
                                                    .createSequentialGroup()
                                                            .addComponent(


            lblNumberOfDeadlocks,
            GroupLayout.PREFERRED_SIZE,
            187,
            GroupLayout.PREFERRED_SIZE).addPreferredGap(
                                                    ComponentPlacement.RELATED)
                                                    .addComponent(
                                                                    textField_2,
            GroupLayout.PREFERRED_SIZE,
                    57,
                    GroupLayout.PREFERRED_SIZE)).addComponent(
                            list, GroupLayout.DEFAULT_SIZE,
                                    GroupLayout.DEFAULT_SIZE,
                                    Short.MAX_VALUE)).addGap(22).addGroup(
            gl_panel_1.createParallelGroup(Alignment.LEADING)
                    .addComponent(lblDetailOfDeadlock,
    GroupLayout.PREFERRED_SIZE, 294, GroupLayout.PREFERRED_SIZE)
                    .addComponent(textArea, GroupLayout.PREFERRED_SIZE,
                    438, GroupLayout.PREFERRED_SIZE)

                    .addComponent(lblSites))
       .addGap(25)));
              gl_panel_1.setVerticalGroup(gl_panel_1
       .createParallelGroup(Alignment.LEADING).addGroup(
                                                   gl_panel_1
       .createSequentialGroup()
       .addContainerGap().addGroup(
       gl_panel_1.createParallelGroup(Alignment.BASELINE)
              .addComponent(lblNumberOfDeadlocks,
                     GroupLayout.PREFERRED_SIZE, 27,
                            GroupLayout.PREFERRED_SIZE)
                                   .addComponent(
                                          textField_2,
       GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
              GroupLayout.PREFERRED_SIZE).addComponent(lblSites))
       .addGap(22).addGroup(gl_panel_1.createParallelGroup(
              Alignment.BASELINE).addComponent(lblListOfFiles)
                     .addComponent(lblDetailOfDeadlock))
                            .addGap(18).addGroup(gl_panel_1
       .createParallelGroup(
              Alignment.BASELINE)
                     .addComponent(list,
                                   GroupLayout.DEFAULT_SIZE,
                                   439,
                                   Short.MAX_VALUE).addComponent(
                                          textArea,
                                          GroupLayout.DEFAULT_SIZE,
                                          442,
                                          Short.MAX_VALUE))
                     .addContainerGap()));
              textArea.setLineWrap(true);
              panel_1.setLayout(gl_panel_1);
       }

       class Task extends SwingWorker<Void, Void> {
              @Override
              protected Void doInBackground() throws Exception {

                     frmDeadlock.setEnabled(false);

                     String sourcePath = textField_1.getText();
                     sourcePath = sourcePath + "\\";

                     DataPreparation   dp   =   new   DataPreparation(new
File(sourcePath));
                     pBar.setValue(75);
                     String exportedDir = dp.getExportedDir();
                     DeadlockDetectionAlgorithm      dda      =      new
DeadlockDetectionAlgorithm(
                                   new       File(sourcePath),       new
File(exportedDir));

                     pBar.setValue(90);
                     display = new DeadlockOutputDisplay(dListModel);
```

```
				display.processGetFileInfo(dListModel);

				pBar.setValue(100);

				textField_2.setText(dListModel.size() + "");
				return null;
			}

			public void done() {

				pBarFrame.dispatchEvent(new WindowEvent(pBarFrame,
						WindowEvent.WINDOW_CLOSING));

				frmDeadlock.setEnabled(true);
				frmDeadlock.toFront();
			}
		}
}
```

B.2.2. DeadlockOutputDisplay.java

The source code of the DeadlockOuputDisplay.java is shown in Table 16.

Table 16 The source code of the DeadlockDetectionDisplay.java

```
package thesis.deadlockdetection.view;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;
import javax.swing.DefaultListModel;

public class DeadlockOutputDisplay {

      public DeadlockOutputDisplay(DefaultListModel<String> dLModel)
{

            String fileName = "D:\\DDReport\\CyclicCondTrue.txt";
            BufferedReader br = null;
            Scanner scanner = null;
            String line = null;

            try {
                  br = new BufferedReader(new FileReader(new
File(fileName)));

                  while ((line = br.readLine()) != null) {
                        line = line

      .substring(line.lastIndexOf("\\") + 1, line.length());
                        dLModel.addElement(line);
                  }

            } catch (Exception e) {

                  e.printStackTrace();

            } finally {

                  if (br != null) {
                        try {

                              br.close();

                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                  }
```

```
              }

        }

        public void processGetFileInfo(DefaultListModel<String>
dLModel) {
              for (int i = 0; i < dLModel.size(); i++) {

                    String filename = dLModel.get(i);

                    File dir = new File("D:\\DDDisplay\\");
                    dir.mkdirs();

                    File file = new File("D:\\DDDisplay\\" +
filename);

                    PrintWriter pw = null;

                    File inputFile = new File("D:\\DDExportedSource\\"
+ filename);

                    BufferedReader br = null;

                    String inputLine = null;

                    File dirOfCond = new File("D:\\DDReport\\");

                    BufferedReader brCond = null;

                    String inputCondLine = null;
                    File[] listAllFiles = dirOfCond.listFiles();

                    int threadCount = 0;

                    try {

                          br = new BufferedReader(new
FileReader(inputFile));

                          pw = new PrintWriter(file);

                          try {

                                while ((inputLine = br.readLine()) !=
null) {

                                      if (inputLine.contains("D:"))
{


      System.out.println(inputLine);
                                            pw.append("File : " +
inputLine + " \n");

                                            System.out.println();
                                            pw.append("\n");
      System.out.println("Deadlock Sites: ");
                                          pw.append("Deadlock
Site: \n");
                                    }
                                    if
(inputLine.contains("synchronized")
                  & !(inputLine.contains("//"))) {
                                    System.out.println(inputLine);
                                          pw.append(inputLine + "
\n");
                                    }
                                    if
(inputLine.contains(".start()")) {
                                          threadCount++;
                                    }
                              }
                              System.out.println();
                              pw.append("\n");
                              System.out.println("Numbers of Thread
= " + threadCount
                                          + "threads.");
                              pw.append("Numbers of Thread = " +
threadCount
                                          + " threads. \n");
                              System.out.println();
                              pw.append("\n");
                              System.out.println("Type(s) of
Deadlock Condition:");
                              pw.append("Type(s) of Deadlock
Condition: \n");

                              for (File Condfile : listAllFiles) {
                                    if (Condfile.isFile()
                                          &
!Condfile.getName().equals(

      "ThreadCountReport.txt")
            & !Condfile.getName().equals(
                                    "EsCondTrueTemp.txt")) {
                                          brCond = new
BufferedReader(
                                                      new
FileReader(Condfile));
                                          String fileCondLine =
null;
                                          while ((fileCondLine =
brCond.readLine()) != null) {
                                                if
(fileCondLine.contains(filename)) {
                                                      if
(Condfile.getName().equals(
                              "AliasingCondTrue.txt")) {
                                                System.out
      .println("Aliasing Condition");
                        pw.append("Aliasing Condition \n");
                                                      } else if
(Condfile.getName().equals(
                              "CyclicCondTrue.txt")) {

      System.out.println("Cyclic Condition");
                                    pw.append("Cyclic Condition
\n");
                                                      } else if
(Condfile.getName().equals("EsCondTrue.txt")) {
                  System.out.println("Escaping Condition");
      pw.append("Escaping Condition \n");
                                                      } else if
(Condfile.getName().equals("NonGuardedCondTrue.txt")) {
                  System.out.println("Non-Guarded Condition");
      pw.append("Non-Guarded Condition \n");
                                                      } else if
(Condfile.getName().equals("ParallelCondTrue.txt")) {
                  System.out.println("Parallel Condition");
      pw.append("Parallel Condition \n");
                                                      } else if
(Condfile.getName().equals("ReachableConTrue.txt")) {
                  System.out.println("Reachable Condition");
      pw.append("Reachable Condition \n");
                                                      } else if
(Condfile.getName().equals("SuperflousCondTrueFile.txt")) {
                  System.out.println("Superflous Condition");
      pw.append("Superflous Condition \n");
                                                      }
                                                }
                                          }
                                    }
                              }
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                  } catch (FileNotFoundException e) {

                        e.printStackTrace();
                  } finally {
                        if (br != null) {
                              try {
                                    br.close();
                              } catch (IOException e) {

                                    e.printStackTrace();
                              }
                        }
                        if (pw != null) {
                              pw.close();
                        }
                  }
            }
      }
}
```

B.3. Controller Package

The source code of the DataPreparation.java is in the thesis.deadlockdetection.controller package. It is shown in Table 17.

Table 17 The source code of the DataPreparation.java

```
package thesis.deadlockdetection.controller;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

public class DataPreparation {

	public ArrayList<File> sourceCodeFiles = new ArrayList<File>();
	public String exportedDir = "";
	public DataPreparation(File inputDir) {

		File[] dir = inputDir.listFiles();
		System.out.println("--- getFiles() ---");
		getFiles(dir);

		System.out.println("--- process exporting multithreading 
API source code ---");
		exportFiles(sourceCodeFiles);
	}

	public void setExportedDir(String exportedDir) {
		this.exportedDir = exportedDir;
	}

	public String getExportedDir() {
		return exportedDir;
	}

	private void exportFiles(ArrayList<File> sourceCodeFiles) {

		File exportedSourceFolder = new 
File("D:\\DDExportedSource\\");
		exportedSourceFolder.mkdirs();
		setExportedDir(exportedSourceFolder.getAbsolutePath());
		System.out.println("--- creating "
					+ exportedSourceFolder.getAbsolutePath() + 
" directory ---");

		for (int i = 0; i < sourceCodeFiles.size(); i++) {

			File file = new
```

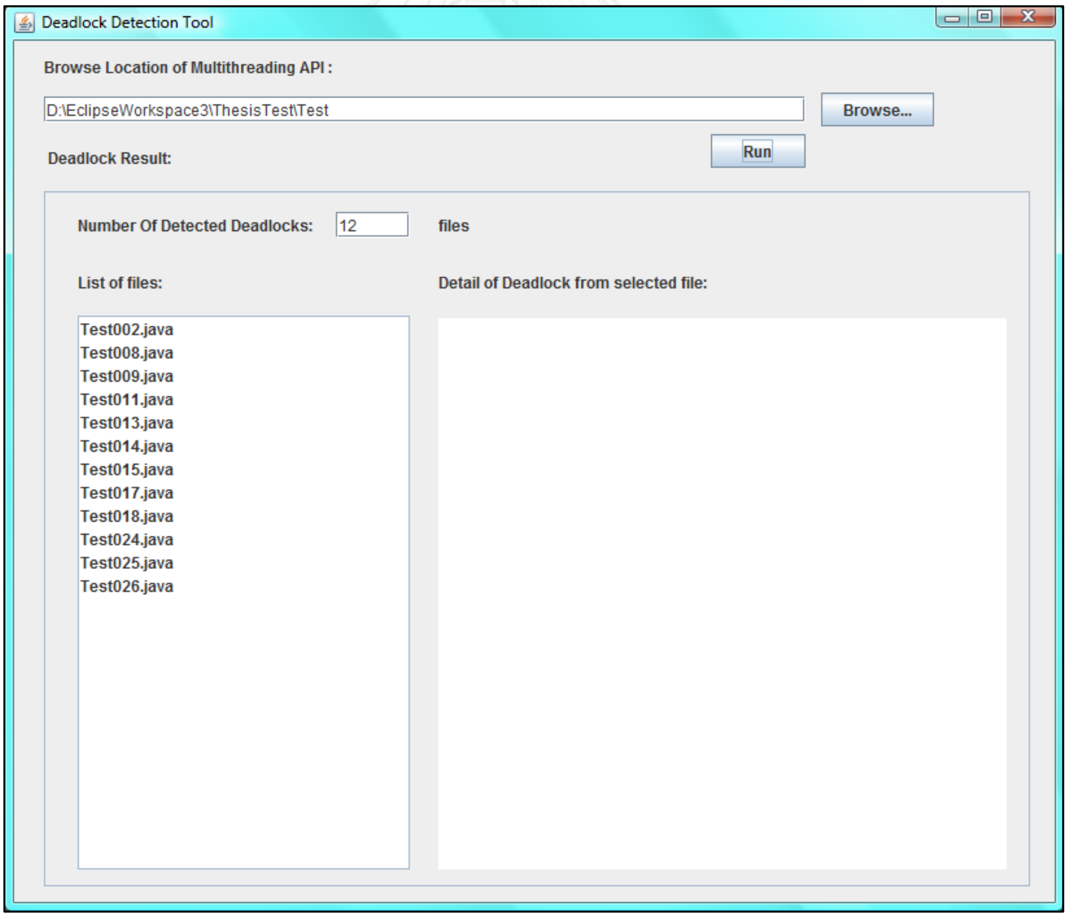
```
File(sourceCodeFiles.get(i).getAbsolutePath());
                    String newExportedFile = "D:\\DDExportedSource\\"
+ file.getName();

                    File destFile = new File(newExportedFile);
                    PrintWriter pw = null;
                    try {

                          pw = new PrintWriter(new
FileWriter(destFile));

                    } catch (IOException e) {

                          e.printStackTrace();
                    }

                    pw.println(file.getAbsolutePath());
                    System.out.println("--- write file " +
file.getAbsolutePath()+ " ---");
                    BufferedReader br = null;
                    String line = "";
                    int lineNumber = 0;

                    try {
                          br = new BufferedReader(new
FileReader(file));
                          try {
                                while ((line = br.readLine()) !=
null) {
                                      lineNumber++;
                                      if (line.contains("//*") |
line.contains("*//")) {
                                            continue;
                                      } else if
(line.contains("extends Thread")) {

                                            if
((line.contains("public")
                                                        &
line.contains("class") & !line.contains("final"))
| (line.contains("class")
             & line.contains("final") & !line.contains("public"))) {
             pw.println("line: " + lineNumber + " :\t\t" + line);
              System.out.println("line: " + lineNumber + " :\t\t" +
line);
                                            } else if
(line.contains("class") & !line.contains("public") &
!line.contains("final")) {
      pw.println("line: " + lineNumber + " :\t\t" + line);
             System.out.println("line: " + lineNumber + " :\t\t" +
line);
                                            } else if
(line.contains("public")
             & line.contains("class") & line.contains("final")) {
             pw.println("line: " + lineNumber + " :\t\t" + line);
              System.out.println("line: " + lineNumber + " :\t\t" +
line);
                                               }
                                         } else if
(line.contains("implements Runnable")) {

                                               if
(line.contains("public")
                                                           &
!line.contains("final") & !line.contains("class") &
!line.contains("interface")) {
             pw.println("line: " + lineNumber + " :\t\t" + line);
             System.out.println("line: " + lineNumber + " :\t\t" +
line);
                                               } else if
((line.contains("class") | line.contains("interface"))  &
!line.contains("public") & !line.contains("final")) {
      pw.println("line: " + lineNumber + " :\t\t" + line);
      System.out.println("line: " + lineNumber + " :\t\t" + line);
                                               } else if
(line.contains("public") & line.contains("final") &
(line.contains("class") | line.contains("interface"))) {
      pw.println("line: " + lineNumber + " :\t\t" + line);
             System.out.println("line: " + lineNumber + " :\t\t" +
line);
                    }
             } else if (line.contains("interface") &
line.contains("extends Thread")) {
                                               pw.println("line: " +
lineNumber + " :\t\t" + line);

      System.out.println("line: " + lineNumber + " :\t\t" + line);
             } else if (line.contains("class") &
(line.contains("extends Thread") | line.contains("implements
Runnable"))) {
                                               pw.println("line: " +
lineNumber + " :\t\t" + line);

      System.out.println("line: " + lineNumber + " :\t\t" + line);
                                         } else if
(line.contains("synchronized")) {
                                               pw.println("line: " +
lineNumber + " :\t\t" + line);

      System.out.println("line: " + lineNumber + " :\t\t" + line);
                                         } else if
(line.contains(".run()")) {
      pw.println("line: " + lineNumber + " :\t\t" + line);
      System.out.println("line: " + lineNumber + " :\t\t" + line);
                                         } else if
(line.contains(".start()")) {
                                               pw.println("line: " +
lineNumber + " :\t\t" + line);

      System.out.println("line: " + lineNumber + " :\t\t" + line);
                                  } else if (line.contains("if (")
      | line.contains("if(") | line.contains("else if (")
        | line.contains("else if(") | line.contains("else")
             & (!line.contains("//") | !line.contains("//*") | !line
                    .contains("*//"))) {
      pw.println("line: " + lineNumber + " :\t\t" + line);
      System.out.println("line: " + lineNumber + " :\t\t" + line);
                                    } else if (line.contains("()
{")) {
                                          pw.println("line: " +
lineNumber + " :\t\t" + line);

      System.out.println("line: " + lineNumber + " :\t\t"
                                                + line);
                                    } else if (line.contains(")
{")) {
                                          pw.println("line: " +
lineNumber + " :\t\t" + line);

      System.out.println("line: " + lineNumber + " :\t\t" + line);
                                    } else if (line.contains("}"))
{
                                          pw.println("line: " +
lineNumber + " :\t\t" + line);

      System.out.println("line: " + lineNumber + " :\t\t" + line);
                                    } else if
(line.contains(".wait(")
                                                       |
line.contains("wait()") | line.contains(".wait()")) {
                                          pw.println("line: " +
lineNumber + " :\t\t" + line);

      System.out.println("line: " + lineNumber + " :\t\t" + line);
      } else if (line.contains("notify()") |
line.contains("notifyAll()")) {
      pw.println("line: " + lineNumber + " :\t\t" + line);
      System.out.println("line: " + lineNumber + " :\t\t" + line);
                                    } else if
(line.contains("sleep(")) {
       pw.println("line: " + lineNumber + " :\t\t" + line);
       System.out.println("line: " + lineNumber + " :\t\t" + line);
                                    } else if (line.contains("new
Thread")) {
       pw.println("line: " + lineNumber + " :\t\t" + line);

      System.out.println("line: " + lineNumber + " :\t\t"
                                                + line);
                                    } else {
                                          if
(line.contains("interface")
                                          ) {
                                                pw.println("line:
" + lineNumber + " :\t\t"+ line);

      System.out.println("line: " + lineNumber+ " :\t\t" + line);
                                          } else if
(line.contains("class")
                                pw.println("line:
" + lineNumber + " :\t\t" + line);

    System.out.println("line: " + lineNumber
                                        + "
:\t\t" + line);
                            } else if
(line.contains("synchronized")) {
                                pw.println("line:
" + lineNumber + " :\t\t"
                                        +
line);

    System.out.println("line: " + lineNumber + " :\t\t" + line);
        } else if (line.contains(".run()")) {
            pw.println("line: " + lineNumber + " :\t\t" +
line);
            System.out.println("line: " + lineNumber + "
:\t\t" + line);
        } else if (line.contains(".start()")) {
            pw.println("line: " + lineNumber + " :\t\t" +
line);
            System.out.println("line: " + lineNumber  + "
:\t\t" + line);
                            } else if
(line.contains("if (") | line.contains("if(") | line.contains("else if
(") | line.contains("else if(") | line.contains("else") &
(!line.contains("//")
 !line.contains("//*") | !line.contains("*//"))) {
    pw.println("line: " + lineNumber + " :\t\t" + line);
    System.out.println("line: " + lineNumber + " :\t\t" + line);
                            } else if
(line.contains("() {")) {
                                pw.println("line:
" + lineNumber + " :\t\t" + line);
     System.out.println("line: " + lineNumber + " :\t\t" + line);
                            } else if
(line.contains(") {")) {
    pw.println("line: " + lineNumber + " :\t\t" + line);
    System.out.println("line: " + lineNumber + " :\t\t" + line);
    } else if (line.contains("}")) {
        pw.println("line: " + lineNumber + " :\t\t"  + line);
        System.out.println("line: " + lineNumber + " :\t\t" +
line);
    } else if (line.contains(".wait(") | line.contains("wait()") |
line.contains(".wait()")) {
        pw.println("line: " + lineNumber + " :\t\t" + line);
        System.out.println("line: " + lineNumber + " :\t\t" +
line);
    } else if (line.contains("notify()") |
line.contains("notifyAll()")) {
    pw.println("line: " + lineNumber + " :\t\t" + line);
    System.out.println("line: " + lineNumber + " :\t\t" + line);
    } else if (line.contains("sleep(")) {
        pw.println("line: " + lineNumber + " :\t\t" + line);
        System.out.println("line: " + lineNumber + " :\t\t" +
line);
      } else if (line.contains("new Thread")) {
            pw.println("line: " + lineNumber + " :\t\t" + line);
             System.out.println("line: " + lineNumber + " :\t\t" +
line);
      } else if (line.contains("=")) {
            pw.println("line: " + lineNumber + " :\t\t" + line);
            System.out.println("line: " + lineNumber + " :\t\t" +
line);
                                          }
                                    }
                              }
                        } catch (IOException e) {

                              e.printStackTrace();

                        }
                  } catch (FileNotFoundException e) {

                        e.printStackTrace();

                  } finally {

                        if (br != null) {

                              try {
                                    br.close();
                              } catch (IOException e) {

                                    e.printStackTrace();
                              }
                        }
                        if (pw != null) {
                              pw.close();
                        }
                  }
            }
      }

      private void getFiles(File[] inputDir) {

            for (int i = 0; i < inputDir.length; i++) {

                  if (inputDir[i].isDirectory()) {
                        File[] tempDir = inputDir[i].listFiles();
                        getFiles(tempDir);

                  } else {

                        String sourceFileName =
inputDir[i].getAbsolutePath();

                        if (sourceFileName.contains(".java")) {
                              sourceCodeFiles.add(inputDir[i]);
                        }
                  }
            }
      }
}
```

APPENDIX C
RESULT OF THE DEADLOCK DETECTION TOOL

The following section of the APPENDIX C shows results of the Deadlock Detection Tool execution and detail of the result.

C.1. Found Deadlocks

Figure 40 shows the screenshot of the Deadlock Detection Tool. Deadlock is found in 12 files that are shown in the list box under the "List of files". 12 files that are found deadlock that are Test002.java, Test008.java, Test009.java, Test011.java, Test013.java, Test014.java, Test015.java, Test017.java, Test018.java, Test024.java, Test025.java and Test026.java.



Browse Location of Multithreading API : D:\EclipseWorkspace3\ThesisTest\Test

Deadlock Result:

Number Of Detected Deadlocks: 12 files

List of files:

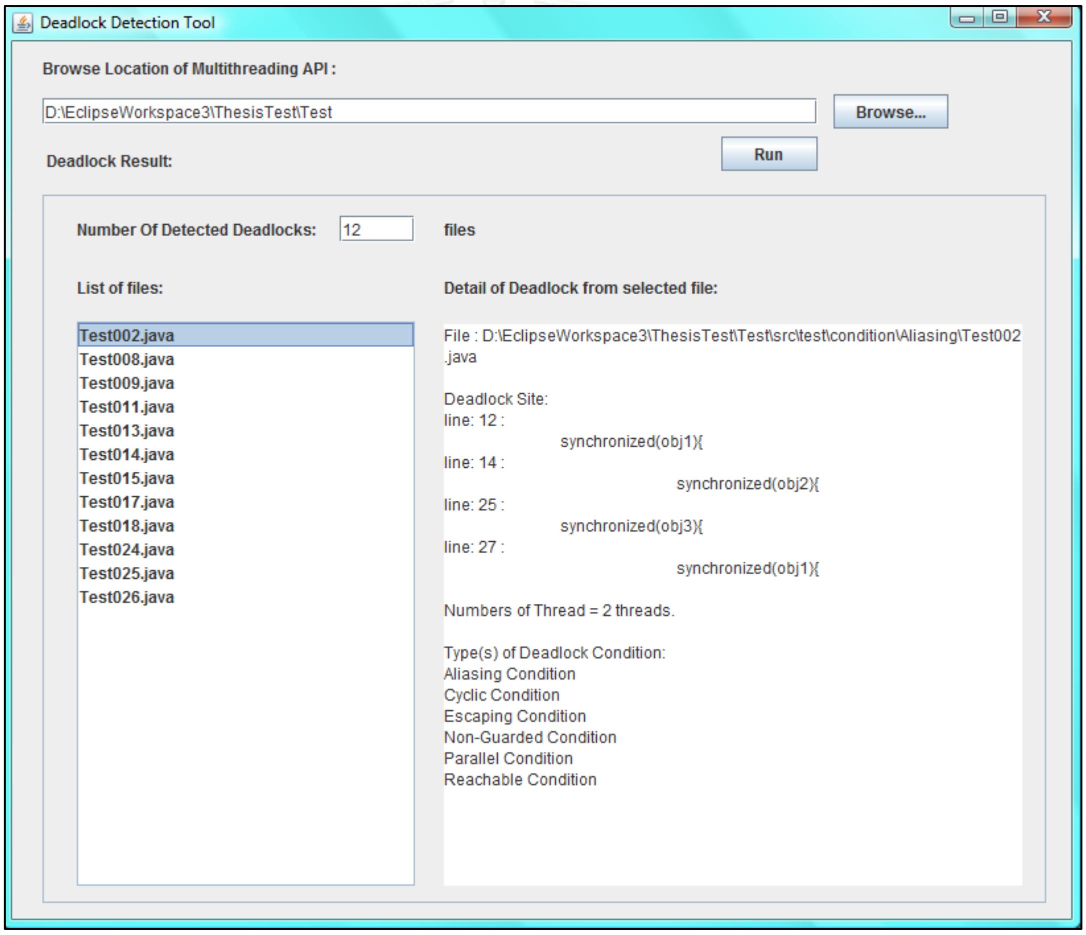
Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

Detail of Deadlock from selected file:

Figure 40 The screenshot of found deadlock

C.2. Detail of Deadlock from Test002.java

After clicking Test002.java in the list box of List of files, the detail of deadlock of Test002.java shows in the text area. Test002.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Aliasing\Test002.java. The deadlock site is at line 12, line 14, line 25 and line 27. There are 2 Threads created. The first Thread locks Obj1 and Obj2 sequentially. The second Thread locks Obj3 and Obj1 sequentially. The deadlock conditions are the Aliasing Condition, the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. Deadlock is the Aliasing Condition therefore Obj2 and Obj3 are aliasing. The screenshot is shown in Figure 41.


Deadlock Detection Tool
Browse Location of Multithreading API :
Browse...
Run
Deadlock Result:
Number Of Detected Deadlocks:
files
List of files:
Detail of Deadlock from selected file:

D:\EclipseWorkspace3\ThesisTest\Test

Number Of Detected Deadlocks: 12 files

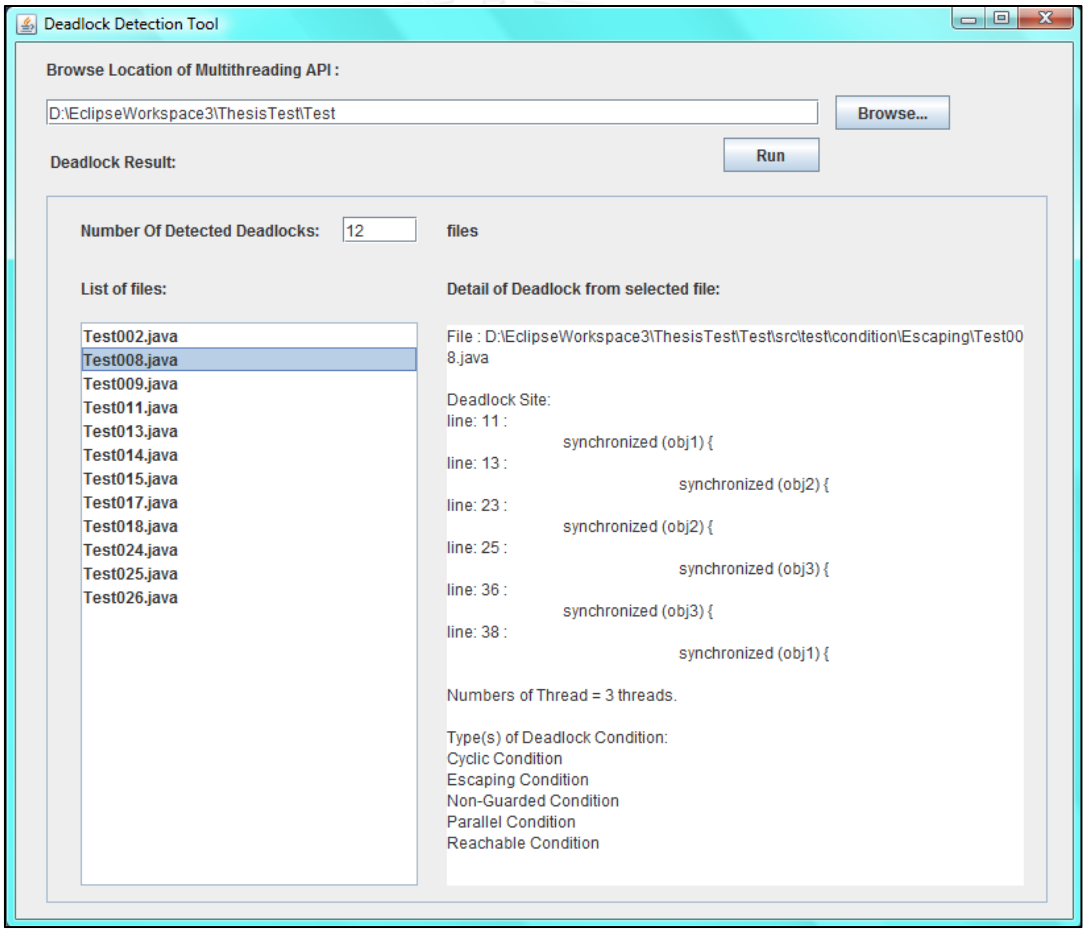
Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Aliasing\Test002
.java

Deadlock Site:
line: 12 :
            synchronized(obj1){
line: 14 :
                        synchronized(obj2){
line: 25 :
            synchronized(obj3){
line: 27 :
                        synchronized(obj1){

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Aliasing Condition
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 41 The screenshot of the detail of deadlock from Test002.java

C.3. Detail of Deadlock from Test008.java

After clicking Test008.java in the list box of List of files, the detail of deadlock of Test008.java shows in the text area. Test008.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Escaping\Test008.java. The deadlock site is at line 11, line 13, line 23, line 25, line 36 and line 38. There are 3 Threads created. The first Thread locks Obj1 and Obj2 sequentially. The second Thread locks Obj2 and Obj3 sequentially. The thrid Thread locks Obj3 and Obj1 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 42.



D:\EclipseWorkspace3\ThesisTest\Test

Number Of Detected Deadlocks: 12 files

List of files:

Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

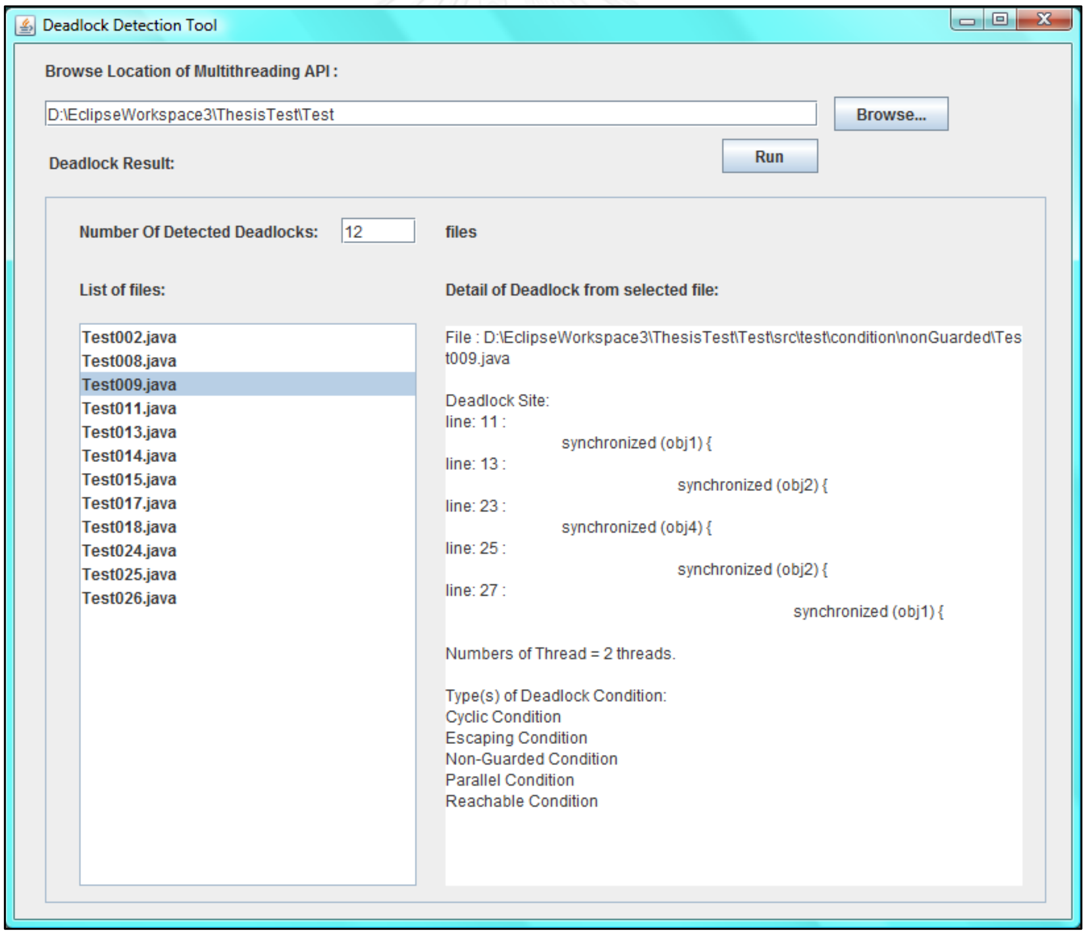
Detail of Deadlock from selected file:

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Escaping\Test00
8.java

Deadlock Site:
line: 11 :
            synchronized (obj1) {
line: 13 :
                        synchronized (obj2) {
line: 23 :
            synchronized (obj2) {
line: 25 :
                        synchronized (obj3) {
line: 36 :
            synchronized (obj3) {
line: 38 :
                        synchronized (obj1) {

Numbers of Thread = 3 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 42 The screenshot of the detail of deadlock from Test008.java

C.4. Detail of Deadlock from Test009.java

After clicking Test009.java in the list box of List of files, the detail of deadlock of Test009.java shows in the text area. Test009.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\nonGuarded\Test009.java. The deadlock site is at line 11, line 13, line 23, line 25 and line 27. There are 2 Threads created. The first Thread locks Obj1 and Obj2 sequentially. The second Thread locks Obj4, Obj2 and Obj1 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 43.



Browse Location of Multithreading API :

D:\EclipseWorkspace3\ThesisTest\Test

Deadlock Result:

Number Of Detected Deadlocks: 12 files

List of files:

Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

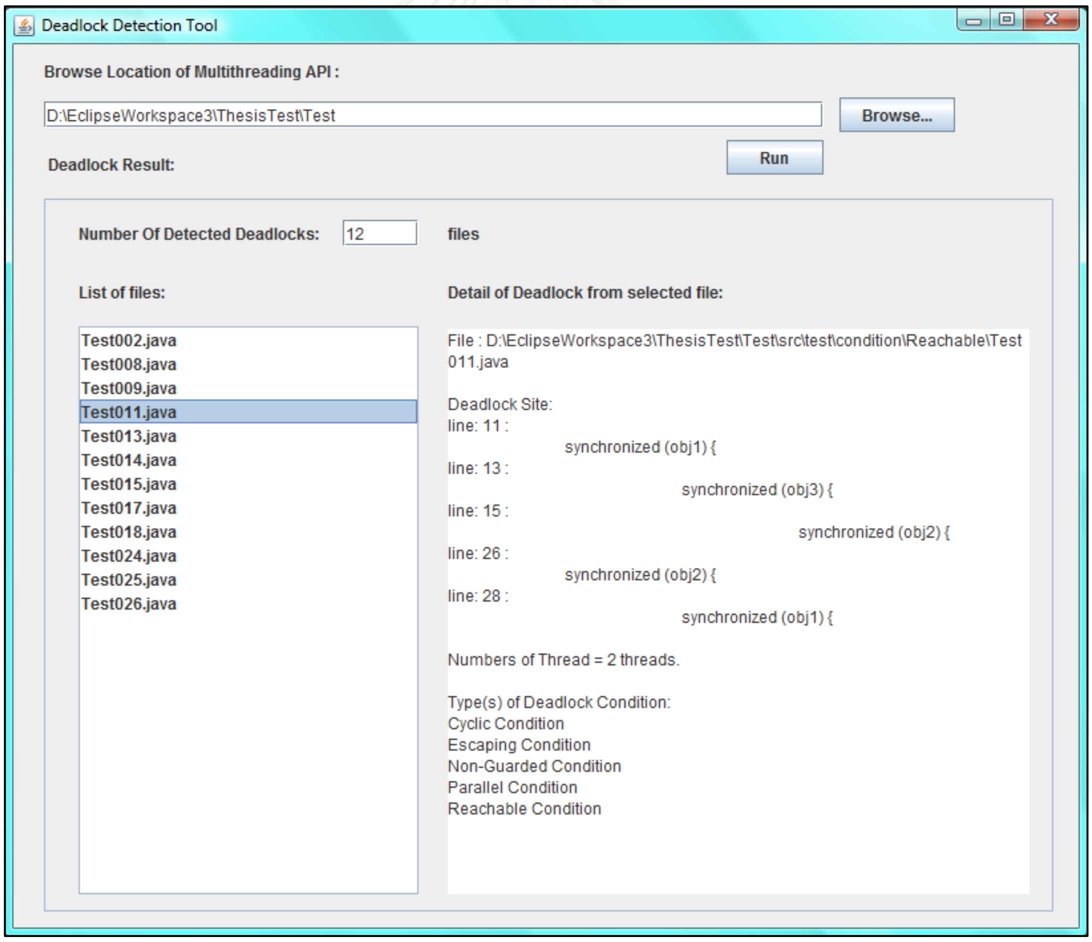
Detail of Deadlock from selected file:

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\nonGuarded\Tes
t009.java

Deadlock Site:
line: 11 :
                synchronized (obj1) {
line: 13 :
                                synchronized (obj2) {
line: 23 :
                synchronized (obj4) {
line: 25 :
                                synchronized (obj2) {
line: 27 :
                                                synchronized (obj1) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 43 The screenshot of the detail of deadlock from Test009.java

C.5. Detail of Deadlock from Test011.java

After clicking Test011.java in the list box of List of files, the detail of deadlock of Test011.java shows in the text area. Test011.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Reachable\Test011.java. The deadlock site is at line 11, line 13, line 15, line 26 and line 28. There are 2 Threads created. The first Thread locks Obj1, Obj3 and Obj2 sequentially. The second Thread locks Obj2 and Obj1 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 44.


Deadlock Detection Tool
Browse Location of Multithreading API :
Browse...
Run
Deadlock Result:
Number Of Detected Deadlocks:
files
List of files:
Detail of Deadlock from selected file:

D:\EclipseWorkspace3\ThesisTest\Test

Number Of Detected Deadlocks: 12 files

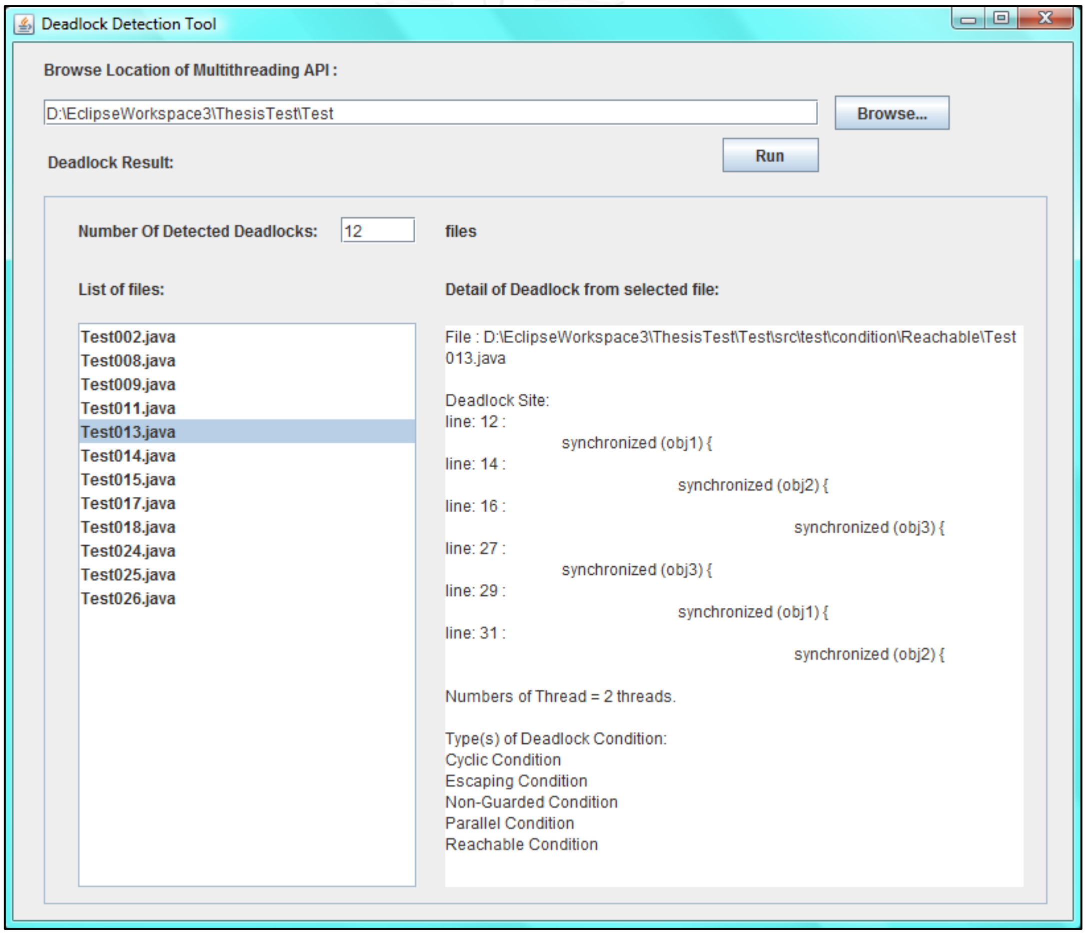
List of files:
Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Reachable\Test
011.java

Deadlock Site:
line: 11 :
                synchronized (obj1) {
line: 13 :
                                synchronized (obj3) {
line: 15 :
                                                synchronized (obj2) {
line: 26 :
                synchronized (obj2) {
line: 28 :
                                synchronized (obj1) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 44 The screenshot of the detail of deadlock from Test011.java

C.6. Detail of Deadlock from Test013.java

After clicking Test013.java in the list box of List of files, the detail of deadlock of Test013.java shows in the text area. Test013.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Reachable\Test013.java. The deadlock site is at line 12, line 14, line 16, line 27, line 29 and line 31. There are 2 Threads created. The first Thread locks Obj1, Obj2 and Obj3 sequentially. The second Thread locks Obj3, Obj1 and Obj2 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 45.



Deadlock Detection Tool
Browse Location of Multithreading API :
Browse...
Run
Deadlock Result:
files
List of files:
Detail of Deadlock from selected file:

Browse Location of Multithreading API : D:\EclipseWorkspace3\ThesisTest\Test

Number Of Detected Deadlocks: 12 files

List of files:

Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

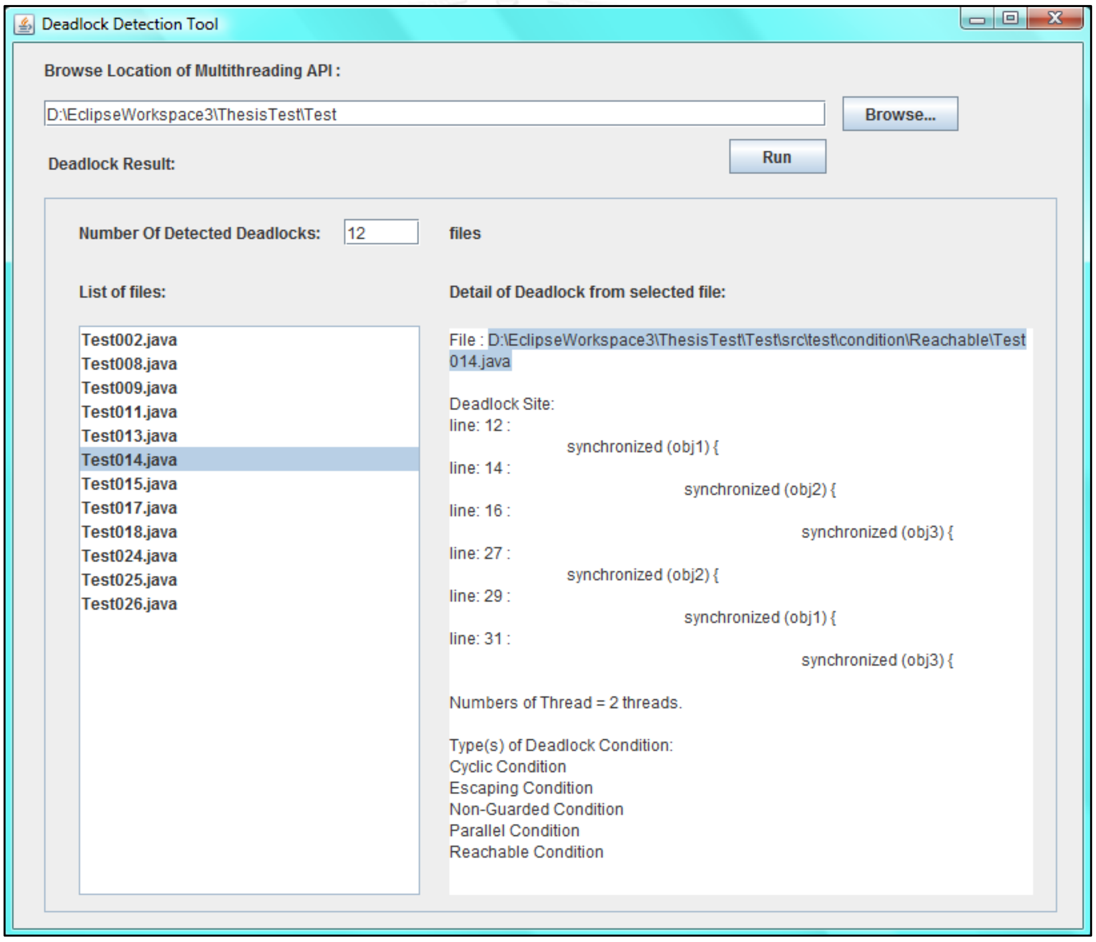
Detail of Deadlock from selected file:

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Reachable\Test
013.java

Deadlock Site:
line: 12 :
                synchronized (obj1) {
line: 14 :
                                synchronized (obj2) {
line: 16 :
                                                synchronized (obj3) {
line: 27 :
                synchronized (obj3) {
line: 29 :
                                synchronized (obj1) {
line: 31 :
                                                synchronized (obj2) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 45 The screenshot of the detail of deadlock from Test013.java

C.7. Detail of Deadlock from Test014.java

After clicking Test014.java in the list box of List of files, the detail of deadlock of Test014.java shows in the text area. Test014.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Reachable\Test014.java. The deadlock site is at line 12, line 14, line 16, line 27, line 29 and line 31. There are 2 Threads created. The first Thread locks Obj1, Obj2 and Obj3 sequentially. The second Thread locks Obj2, Obj1 and Obj3 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 46.


Deadlock Detection Tool
Browse Location of Multithreading API :
Browse...
Run
Deadlock Result:
List of files:
Detail of Deadlock from selected file:

D:\EclipseWorkspace3\ThesisTest\Test

Number Of Detected Deadlocks: 12 files

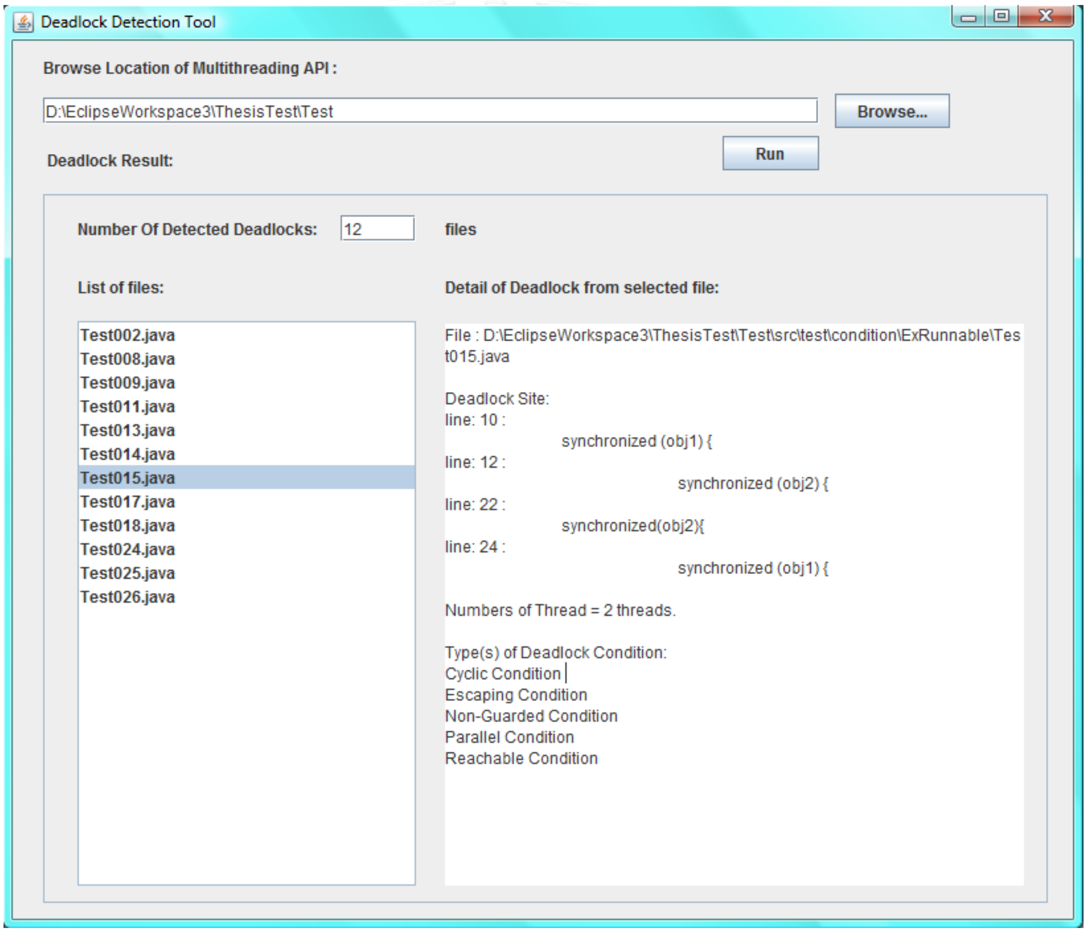
Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Reachable\Test
014.java

Deadlock Site:
line: 12 :
            synchronized (obj1) {
line: 14 :
                        synchronized (obj2) {
line: 16 :
                                    synchronized (obj3) {
line: 27 :
            synchronized (obj2) {
line: 29 :
                        synchronized (obj1) {
line: 31 :
                                    synchronized (obj3) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 46 The screenshot of the detail of deadlock from Test014.java

C.8. Detail of Deadlock from Test015.java

After clicking Test015.java in the list box of List of files, the detail of deadlock of Test015.java shows in the text area. Test015.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\ExRunnable\Test015.java. The deadlock site is at line 10, line 12, line 22 and line 24. There are 2 Threads created. The first Thread locks Obj1 and Obj2 sequentially. The second Thread locks Obj2 and Obj1 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 47.

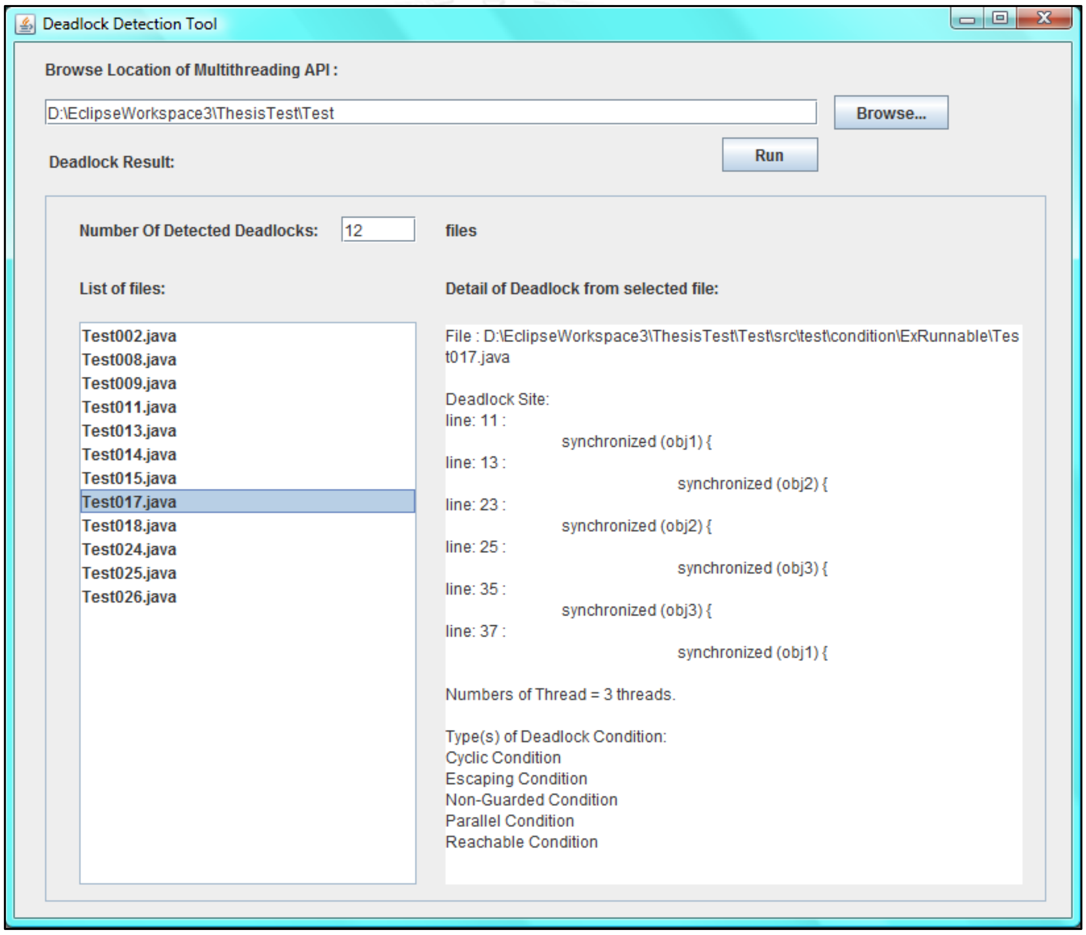


```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\ExRunnable\Tes
t015.java

Deadlock Site:
line: 10 :
                synchronized (obj1) {
line: 12 :
                                synchronized (obj2) {
line: 22 :
                synchronized(obj2){
line: 24 :
                                synchronized (obj1) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 47 The screenshot of the detail of deadlock from Test015.java

C.9. Detail of Deadlock from Test017.java

After clicking Test017.java in the list box of List of files, the detail of deadlock of Test017.java shows in the text area. Test017.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\ExRunnable\Test017.java. The deadlock site is at line 11, line 13, line 23, line 25, line 35 and line 37. There are 3 Threads created. The first Thread locks Obj1 and Obj2 sequentially. The second Thread locks Obj2 and Obj3 sequentially. The third Thread locks Obj3 and Obj1 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 48.



Browse Location of Multithreading API :

D:\EclipseWorkspace3\ThesisTest\Test

Deadlock Result:

Number Of Detected Deadlocks: 12 files

List of files:

Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

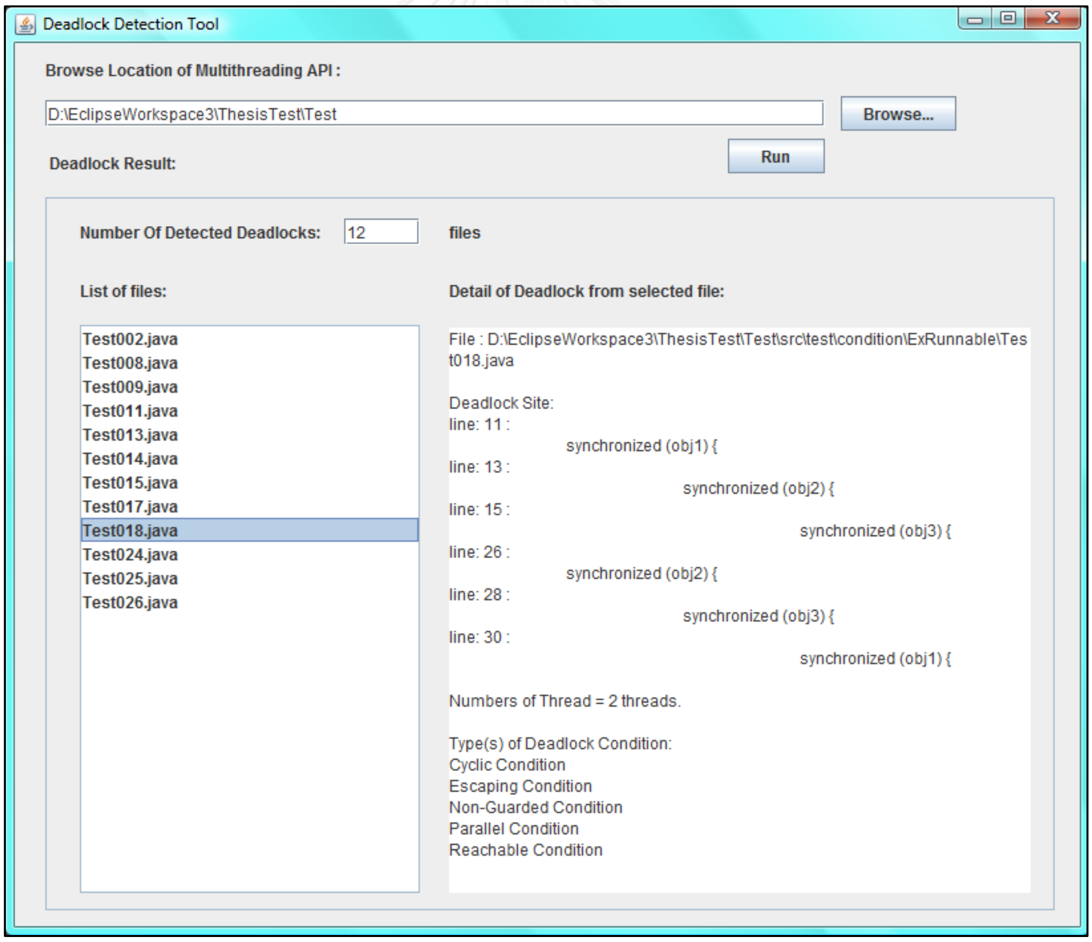
Detail of Deadlock from selected file:

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\ExRunnable\Tes
t017.java

Deadlock Site:
line: 11 :
            synchronized (obj1) {
line: 13 :
                        synchronized (obj2) {
line: 23 :
            synchronized (obj2) {
line: 25 :
                        synchronized (obj3) {
line: 35 :
            synchronized (obj3) {
line: 37 :
                        synchronized (obj1) {

Numbers of Thread = 3 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 48 The screenshot of the detail of deadlock from Test017.java

C.10. Detail of Deadlock from Test018.java

After clicking Test018.java in the list box of List of files, the detail of deadlock of Test018.java shows in the text area. Test018.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\ExRunnable\Test018.java. The deadlock site is at line 11, line 13, line 15, line 26, line 28 and line 30. There are 2 Threads created. The first Thread locks Obj1, Obj2 and Obj3 sequentially. The second Thread locks Obj2, Obj3 and Obj1 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 49.


Deadlock Detection Tool
Browse Location of Multithreading API :
Browse...
Run
Deadlock Result:
List of files:
Detail of Deadlock from selected file:

D:\EclipseWorkspace3\ThesisTest\Test

Number Of Detected Deadlocks: 12 files

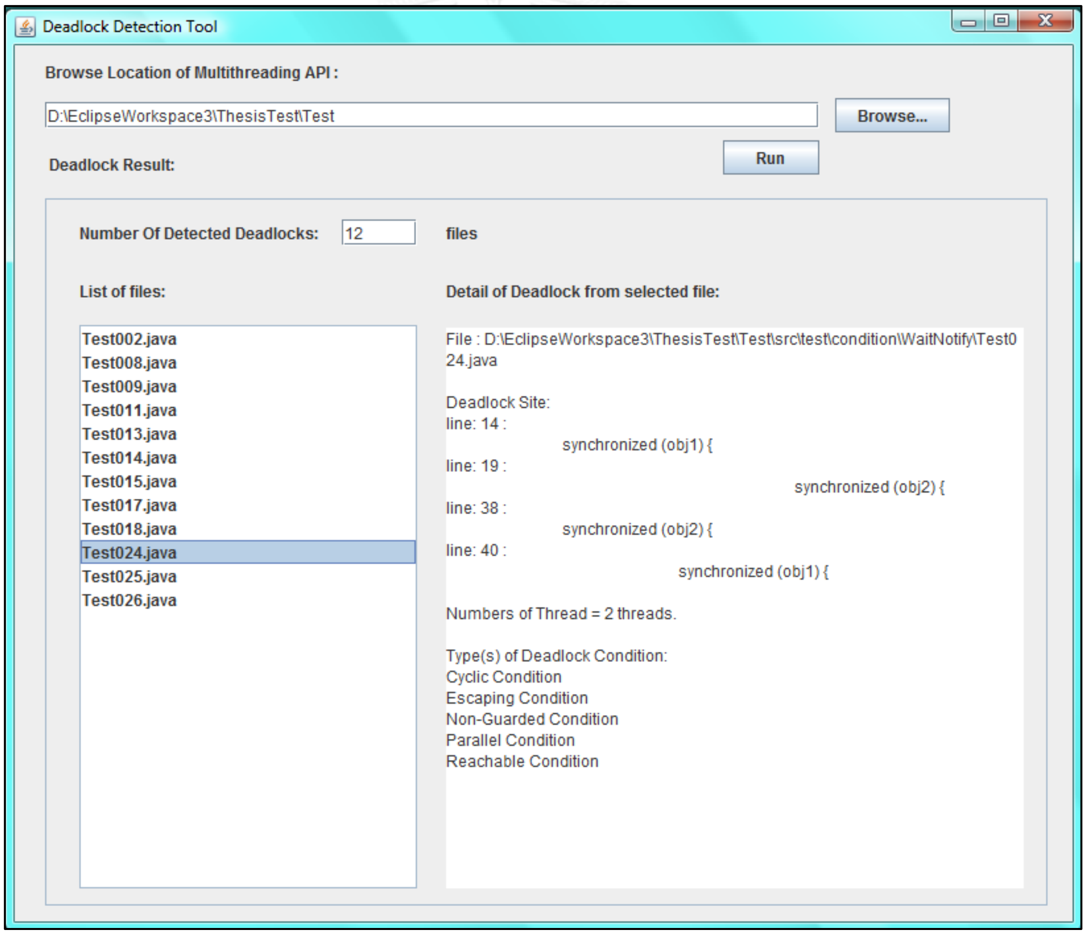
Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\ExRunnable\Tes
t018.java

Deadlock Site:
line: 11 :
                synchronized (obj1) {
line: 13 :
                                synchronized (obj2) {
line: 15 :
                                                synchronized (obj3) {
line: 26 :
                synchronized (obj2) {
line: 28 :
                                synchronized (obj3) {
line: 30 :
                                                synchronized (obj1) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 49 The screenshot of the detail of deadlock from Test018.java

C.11. Detail of Deadlock from Test024.java

After clicking Test024.java in the list box of List of files, the detail of deadlock of Test024.java shows in the text area. Test024.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\WaitNotify\Test024.java. The deadlock site is at line 14, line 19, line 38 and line 40. There are 2 Threads created. The first Thread locks Obj1 and Obj2 sequentially. The second Thread locks Obj2 and Obj1 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 50.

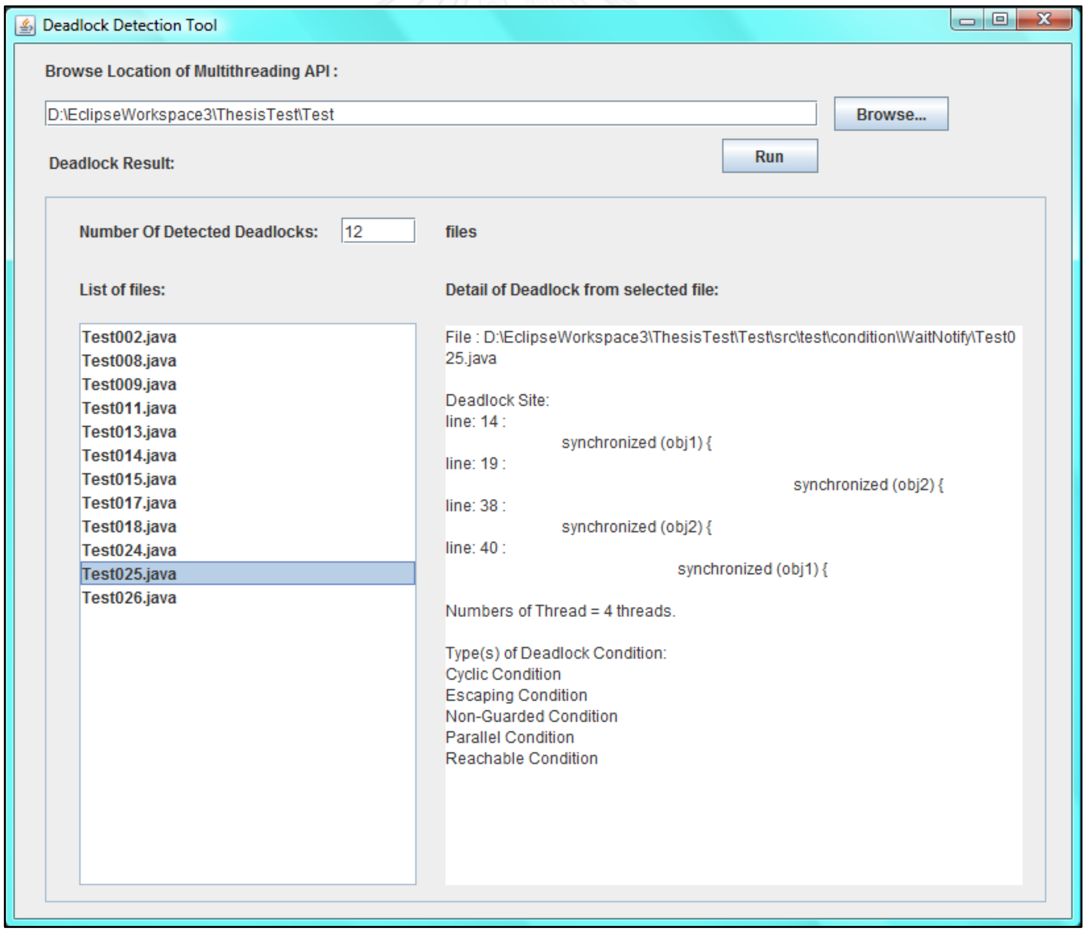


```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\WaitNotify\Test0
24.java

Deadlock Site:
line: 14 :
            synchronized (obj1) {
line: 19 :
                              synchronized (obj2) {
line: 38 :
            synchronized (obj2) {
line: 40 :
                        synchronized (obj1) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 50 The screenshot of the detail of deadlock from Test024.java

C.12. Detail of Deadlock from Test025.java

After clicking Test025.java in the list box of List of files, the detail of deadlock of Test025.java shows in the text area. Test025.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\WaitNotify\Test025.java. The deadlock site is at line 14, line 19, line 38 and line 40. There are 4 Threads created. There are two Threads lock Obj1 and Obj2 sequentially. There are two Threads lock Obj2 and Obj1 sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 51.


Deadlock Detection Tool
Browse Location of Multithreading API :
D:\EclipseWorkspace3\ThesisTest\Test
Browse...
Deadlock Result:
Run
Number Of Detected Deadlocks: 12 files
List of files:
Detail of Deadlock from selected file:

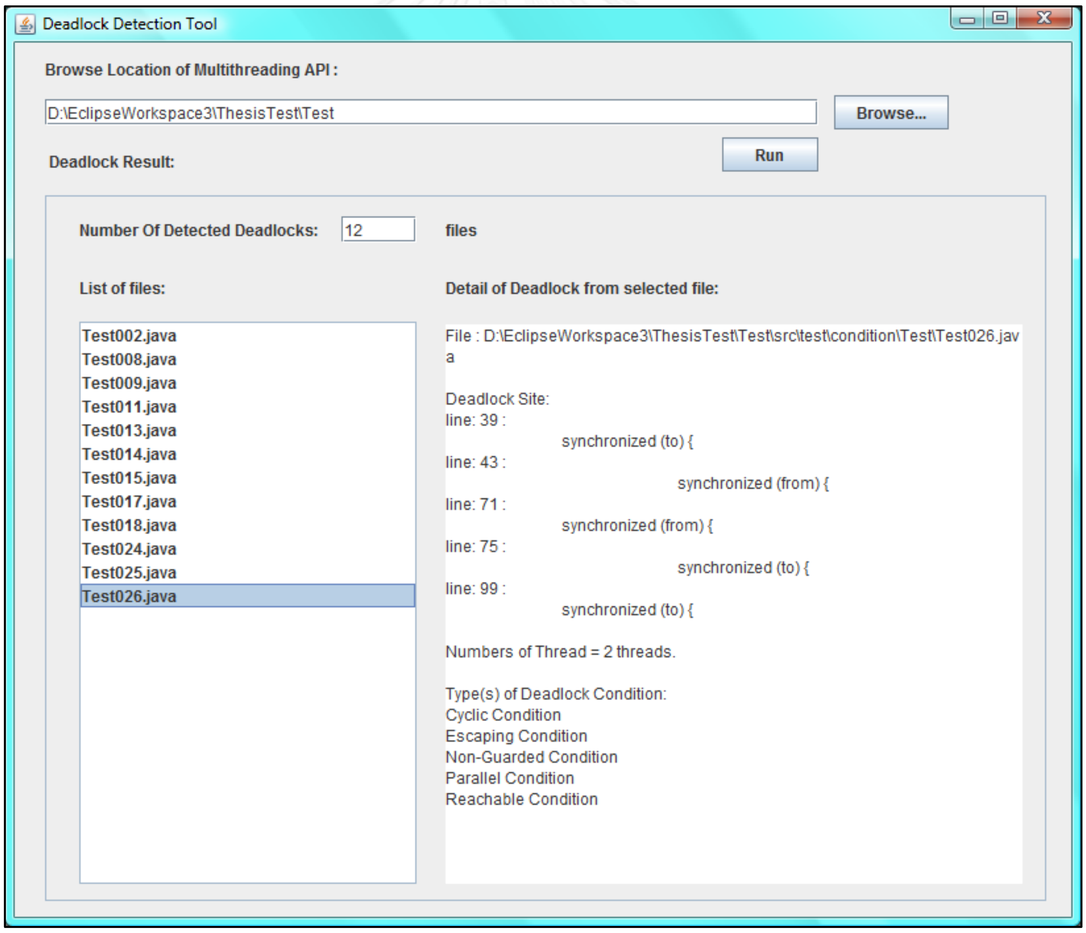
Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\WaitNotify\Test0
25.java

Deadlock Site:
line: 14 :
            synchronized (obj1) {
line: 19 :
                              synchronized (obj2) {
line: 38 :
            synchronized (obj2) {
line: 40 :
                        synchronized (obj1) {

Numbers of Thread = 4 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 51 The screenshot of the detail of deadlock from Test025.java

C.13. Detail of Deadlock from Test026.java

After clicking Test026.java in the list box of List of files, the detail of deadlock of Test026.java shows in the text area. Test026.java file is located at D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Test\Test026.java. The deadlock site is at line 39, line 43, line 75 and line 99. There are 2 Threads created. The first Thread locks Object ‘to’ and Object ‘from’ sequentially. The second Thread locks Object ‘from’ and Object ‘to’ sequentially. The deadlock conditions are the Cyclic Lock Dependency Condition, the Escaping Condition, the Non-Guarded Condition, the Parallel Condition and the Reachable Condition. The screenshot is shown in Figure 52.


Deadlock Detection Tool
Browse Location of Multithreading API :
Browse...
Deadlock Result:
Run
List of files:
Detail of Deadlock from selected file:

D:\EclipseWorkspace3\ThesisTest\Test

Number Of Detected Deadlocks: 12 files

Test002.java
Test008.java
Test009.java
Test011.java
Test013.java
Test014.java
Test015.java
Test017.java
Test018.java
Test024.java
Test025.java
Test026.java

```
File : D:\EclipseWorkspace3\ThesisTest\Test\src\test\condition\Test\Test026.jav
a

Deadlock Site:
line: 39 :
                synchronized (to) {
line: 43 :
                                synchronized (from) {
line: 71 :
                synchronized (from) {
line: 75 :
                                synchronized (to) {
line: 99 :
                synchronized (to) {

Numbers of Thread = 2 threads.

Type(s) of Deadlock Condition:
Cyclic Condition
Escaping Condition
Non-Guarded Condition
Parallel Condition
Reachable Condition
```

Figure 52 The screenshot of the detail of deadlock from Test026.java

APPENDIX D
SOURCE CODE OF TEST EXAMPLES OF MULTITHREADING API

The following are the test files of the multithreading applications for testing the Deadlock Detection Algorithm. There are 26 test files; including 12 test files that have deadlock and 14 test files that are not deadlock. Table 18 shows the the list of test files for testing on each condition.

Table 18 The list of test files for testing on each condition

Conditions	Number of test files	List of test files
Aliasing	5	Test001.java, Test002.java, Test003.java, Test022.java and Test023.java
Cyclic Lock Dependency	12	Test002.java, Test008.java, Test009.java, Test011.java, Test013.java, Test014.java, Test015.java, Test017.java, Test018.java, Test024.java, Test025.java and Test026.java
Escaping	24	Test001.java, Test002.java, Test003.java, Test004.java, Test005.java, Test006.java, Test007.java, Test008.java, Test009.java, Test010.java, Test011.java, Test012.java, Test013.java, Test014.java, Test015.java, Test016.java, Test017.java, Test018.java, Test019.java, Test022.java, Test023.java, Test024.java, Test025.java and Test026.java
Parallel	24	Test001.java, Test002.java, Test003.java, Test004.java, Test005.java, Test006.java, Test007.java, Test008.java, Test009.java, Test010.java, Test011.java, Test012.java, Test013.java, Test014.java, Test015.java, Test016.java, Test017.java, Test018.java, Test019.java, Test022.java, Test023.java, Test024.java Test025.java and Test026.java
Non-Guarded Lock	20	Test001.java, Test002.java, Test004.java, Test005.java, Test006.java, Test007.java, Test008.java, Test009.java, Test011.java, Test013.java, Test014.java, Test015.java, Test017.java, Test018.java, Test021.java, Test022.java, Test023.java, Test024.java, Test025.java and Test026.java
Reachable	23	Test001.java, Test002.java, Test003.java, Test004.java, Test005.java, Test006.java, Test007.java, Test008.java, Test009.java, Test010.java, Test011.java, Test012.java, Test013.java, Test014.java, Test015.java, Test016.java, Test017.java, Test018.java, Test019.java, Test022.java, Test024.java, Test025.java and Test026.java
Superfluous Lock	1	Test001.java

We have 5 files to test on the Aliasing Condition that are Test001.java, Test002.java, Test003.java, Test022.java and Test023.java. We have 12 files to test on the Cyclic Lock Dependency Condition that are Test002.java, Test008.java, Test009.java, Test011.java, Test013.java, Test014.java, Test015.java, Test017.java, Test018.java, Test024.java, Test025.java and Test026.java. We have 24 test files to test on the Escaping Condition that are Test001.java, Test002.java, Test003.java, Test004.java, Test005.java, Test006.java, Test007.java, Test008.java, Test009.java, Test010.java, Test011.java, Test012.java, Test013.java, Test014.java, Test015.java, Test016.java, Test017.java, Test018.java, Test019.java, Test022.java, Test023.java, Test024.java, Test025.java and Test026.java. We have 24 files to test on the Parallel Condition that are Test001.java, Test002.java, Test003.java, Test004.java, Test005.java, Test006.java, Test007.java, Test008.java, Test009.java, Test010.java, Test011.java, Test012.java, Test013.java, Test014.java, Test015.java, Test016.java, Test017.java, Test018.java, Test019.java, Test022.java, Test023.java, Test024.java Test025.java and Test026.java. We have 20 test files to test on the Non-Guarded Lock Condition that are Test001.java, Test002.java, Test004.java, Test005.java, Test006.java, Test007.java, Test008.java, Test009.java, Test011.java, Test013.java, Test014.java, Test015.java, Test017.java, Test018.java, Test021.java, Test022.java, Test023.java, Test024.java, Test025.java and Test026.java. We have 23 test files to test on the Reachable Condition that are Test001.java, Test002.java, Test003.java, Test004.java, Test005.java, Test006.java, Test007.java, Test008.java, Test009.java, Test010.java, Test011.java, Test012.java, Test013.java, Test014.java, Test015.java, Test016.java, Test017.java, Test018.java, Test019.java, Test022.java, Test024.java, Test025.java and Test026.java. And we have 1 file to test on the Superfluous Lock Condition that is Test001.java.

Table 19 shows summary of expected result of test files. x is represented the condition of that file, N is represented deadlock does not occur in that file and Y is represented deadlock occurs in that file.

Table 19 Summary result of test files

Files (.java)	Result	Deadlock	Aliasing	Cyclic Lock Dependency	Escaping	Parallel	Non-Guarded Lock	Reachable	Superfluous Lock	Thread Number
Test001	Expected	N	x		x	x	x	x	x	2
Test002	Expected	Y	x	x	x	x	x	x		2
Test003	Expected	N	x		x	x		x		2
Test004	Expected	N			x	x	x	x		2
Test005	Expected	N			x	x	x	x		2
Test006	Expected	N			x	x	x	x		2
Test007	Expected	N			x	x	x	x		3
Test008	Expected	Y		x	x	x	x	x		3
Test009	Expected	Y		x	x	x	x	x		2
Test010	Expected	N			x	x		x		2
Test011	Expected	Y		x	x	x	x	x		2
Test012	Expected	N			x	x		x		2
Test013	Expected	Y		x	x	x	x	x		2
Test014	Expected	Y		x	x	x	x	x		2
Test015	Expected	Y		x	x	x	x	x		2
Test016	Expected	N			x	x		x		2
Test017	Expected	Y		x	x	x	x	x		3
Test018	Expected	Y		x	x	x	x	x		2
Test019	Expected	N			x	x		x		3
Test020	Expected	N								0
Test021	Expected	N					x			1
Test022	Expected	N	x		x	x	x	x		2
Test023	Expected	N	x		x	x	x			2
Test024	Expected	Y		x	x	x	x	x		2
Test025	Expected	Y		x	x	x	x	x		4
Test026	Expected	Y		x	x	x	x	x		2

The following section elaborates more detail of each test file.

1. Test001.java

Deadlock does not occur in Test001.java file. Test001.java has 2 Threads. We have obj1 and obj2 that are created from the Object Class. Obj1 and obj2 has the same reference so in Java obj1 and obj2 are alias. Thread1 locks obj1 and obj2 sequentially and Thread2 locks obj2 and obj1 sequentially. Both Threads lock the same object therefore there is no deadlock occur in Test001.java. In this example not only match the Aliasing Condition but also the Parallel Condition, the Escaping Condition, the Reachable Condition, the Superfluous Lock Condition and the Non-Guarded Lock Condition. The source code of Test001.java is shown in Table 20.

Table 20 The source code of Test001.java that deadlock does not occur

```
package test.condition.Aliasing;

public class Test001 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj1";

class MyWork1 extends Thread{

		public void run(){
			synchronized(obj1){
				System.out.println("obj1 is locked.");
				synchronized(obj2){
					System.out.println("obj2 is locked.");
				}
			}
			System.out.println("finish work1.");
		}
	}

	class MyWork2 extends Thread{

		public void run(){
			synchronized(obj2){
				System.out.println("obj2 is locked.");
				synchronized(obj1){
					System.out.println("obj1 is locked.");
				}
			}
			System.out.println("finish work2.");
		}
	}
	public static void main(String[] args) {

		Test001 test001 = new Test001();
		Test001.MyWork1 work1 = test001. new MyWork1();
		Test001.MyWork2 work2 = test001. new MyWork2();
		work1.start();
		work2.start();
	}
}
```

2. Test002.java

Deadlock occurs in Test002.java file. Test002.java has 2 Threads. We have obj1, obj2 and obj3 that are created from the Object Class. Obj2 and obj3 are alias. Obj3 is alias to obj2. Thread1 locks obj1 and obj2 sequentially and Thread2 locks obj3 and obj1 sequentially. Both threads lock obj1 and obj2 in reverse order as well. Therefore deadlock conditions that occur in Test002.java are the Aliasing Condition, the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Dependency condition. The source code of Test002.java is shown in Table 21.

Table 21 The source code of Test002.java that deadlock occurs

```
package test.condition.Aliasing;

public class Test002 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj2";
	private Object obj3 = obj2;

class MyWork1 extends Thread{

		public void run(){
			synchronized(obj1){
				System.out.println("obj1 is locked.");
				synchronized(obj2){
					System.out.println("obj2 is locked.");
				}
			}
			System.out.println("finish work1.");
		}
	}
```

```
        class MyWork2 extends Thread{

                public void run(){
                        synchronized(obj3){
                                System.out.println("obj3 is locked.");
                                synchronized(obj1){
                                        System.out.println("obj1 is locked.");
                                }
                        }
                        System.out.println("finish work2.");
                }
        }

        public static void main(String[] args) {

                Test002 test002 = new Test002();
                Test002.MyWork1 work1 = test002. new MyWork1();
                Test002.MyWork2 work2 = test002. new MyWork2();
                work1.start();
                work2.start();

        }
}
```

3. Test003.java

Deadlock does not occur in Test003.java. Test003.java has 2 Threads. We have obj1, obj2 and obj3 that are created from the Object Class. Obj2 is alias to obj1. Thread1 locks obj1 and obj3 sequentially and Thread2 locks obj2 and obj3 sequentially. Since obj1 and obj2 are alias and the lock orders of both Threads are not reverse, the deadlock does not occur. The conditions of Test003.java are the Aliasing Condition, the Parallel Condition, the Escaping Condition, and the Reachable Condition. The source code of Test003.java is shown in Table 22.

Table 22 The source code of Test003.java that deadlock does not occur

```
package test.condition.Aliasing;

public class Test003 {

	private Object obj1 = "obj1";
	private Object obj2 = obj1;
	private Object obj3 = "obj3";

	class MyWork1 extends Thread {

		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				synchronized (obj3) {
					System.out.println("obj3 is locked.");
				}
			}
			System.out.println("finish work1.");
		}
	}

	class MyWork2 extends Thread {

		public void run() {
			synchronized (obj2) {
				System.out.println("obj2 is locked.");
				synchronized (obj3) {
					System.out.println("obj3 is locked.");
				}
			}
			System.out.println("finish work2.");
		}
	}

	public static void main(String[] args) {
		Test003 test003 = new Test003();
		Test003.MyWork1 work1 = test003. new MyWork1();
		Test003.MyWork2 work2 = test003. new MyWork2();
		work1.start();
		work2.start();
	}
}
```

4. Test004.java

Deadlock does not occur in Test004.java. Test004.java has 2 Threads. We have obj1, obj2 and obj3 that are created from the Object Class. Thread1 locks obj1 and obj2 sequentially and Thread2 lock obj3 and obj1 sequentially. The lock orders of both threads are not reverse therefore deadlock does not occur. The conditions of Test004.java are the Parallel Condition, the Escaping Condition, the Reachable Condition and the Non-Guarded Lock Condition. The source code of Test004.java is shown in Table 23.

Table 23 The source code of Test004.java that deadlock does not occur

```
package test.condition.Cyclic;

public class Test004 {

        private Object obj1 = "obj1";
        private Object obj2 = "obj2";
        private Object obj3 = "obj3";

        class MyWork1 extends Thread {

                public void run() {
                        synchronized (obj1) {
                                System.out.println("obj1 is locked.");
                                synchronized (obj2) {
                                        System.out.println("obj2 is locked.");
                                }
                        }
                        System.out.println("finish work1.");
                }
        }

        class MyWork2 extends Thread {

                public void run() {
                        synchronized (obj3) {
                                System.out.println("obj3 is locked.");
                                synchronized (obj1) {
                                        System.out.println("obj1 is locked.");
                                }
                        }
                        System.out.println("finish work2.");
                }
        }
```

```
	public static void main(String[] args) {

		Test004 test004 = new Test004();
		Test004.MyWork1 work1 = test004.new MyWork1();
		Test004.MyWork2 work2 = test004.new MyWork2();
		work1.start();
		work2.start();
	}
}
```

5. Test005.java

Deadlock does not occur in Test005.java. Test005.java has 2 Threads. We have obj1 that is created from the Object Class. Thread1 locks obj1 and 'this' sequentially and Thread2 locks 'this' and lock1 sequentially. The lock orders of both threads are reverse but in Java 'this' locking means lock this class that is already locked when this application is executed. Therefore both threads lock 'this' and obj1 sequentially and does not occur deadlock. The conditions of Test005.java are the Parallel Condition, the Escaping Condition, the Reachable Condition and the Non-Guarded Lock Condition. The source code of Test005.java is shown in Table 24.

Table 24 The source code of Test005.java that deadlock does not occur

```
package test.condition.Aliasing;

public class Test005 {

	Object obj1 = "obj1";

	class Work1 extends Thread {
		public void run() {
			synchronized (this) {
				System.out.println("-this- is locked.");
				synchronized (obj1) {
					System.out.println("obj1 is locked.");
				}
			}
			System.out.println("finish work1.");
		}
	}

	class Work2 extends Thread {
		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				synchronized (this) {
					System.out.println("-this- is locked");
				}
			}
```

```
                        System.out.println("finish work2.");
                }
        }

        public static void main(String[] args) {
                Test005 test005 = new Test005();
                Test005.Work1 work1 = test005.new Work1();
                Test005.Work2 work2 = test005.new Work2();
                work1.start();
                work2.start();
        }
}
```

6. Test006.java

Deadlock does not occur in Test006.java. Test006.java has 2 Threads. We have obj1 and obj2 that are created from the Object Class. Thread1 locks obj1 and obj2 sequentially and Thread2 locks obj2. The conditions of Test006 are the Parallel Condition, the Escaping Condition, the Reachable Condition and the Non-Guarded Lock Condition. The source code of Test006.java is shown in Table 25.

Table 25 The source code of Test006.java that deadlock does not occur

```
package test.condition.Escaping;

public class Test006 {

        private Object obj1 = "obj1";
        private Object obj2 = "obj2";

        class MyWork1 extends Thread {

                public void run() {
                        synchronized (obj1) {
                                System.out.println("obj1 is locked.");
                                synchronized (obj2) {
                                        System.out.println("obj2 is locked");
                                }
                        }
                        System.out.println("finish work1.");
                }
        }

        class MyWork2 extends Thread {

                public void run() {
                        synchronized (obj2) {
                                System.out.println("obj2 is locked.");
                        }
                        System.out.println("finish work2.");
                }
```

```
	}

	public static void main(String[] args) {
		Test006 test006 = new Test006();
		Test006.MyWork1 work1 = test006.new MyWork1();
		Test006.MyWork2 work2 = test006.new MyWork2();
		work1.start();
		work2.start();
	}
}
```

7. Test007.java

Deadlock does not occur in Test007.java. Test007.java has 3 Threads. We have obj1, obj2 and obj3 that are created from the Object Class. Thread1 locks obj1 and obj2 sequentially, Thread2 locks obj2 and Thread3 locks obj1 and obj3 sequentially. Lock orders of three threads are not reverse therefore deadlock does not occur. The conditions of Test007.java are the Parallel Condition, the Escaping Condition, the Reachable Condition and the Non-Guarded Lock Condition. The source code of Test007.java is shows in Table 26.

Table 26 The source code of Test007.java that deadlock does not occur

```
package test.condition.Escaping;

public class Test007 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj2";
	private Object obj3 = "obj3";

	class MyWork1 extends Thread {
		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				synchronized (obj2) {
					System.out.println("obj2 is locked");
				}
			}
			System.out.println("finish work1.");
		}
	}
	class MyWork2 extends Thread {
		public void run() {
			synchronized (obj2) {
				System.out.println("obj2 is locked.");
			}
			System.out.println("finish work2.");
		}
	}

	class MyWork3 extends Thread {
		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				synchronized (obj3) {
					System.out.println("obj3 is locked.");
				}
			}
			System.out.println("finish work3.");
		}
	}

	public static void main(String[] args) {
		Test007 test007 = new Test007();
		Test007.MyWork1 work1 = test007.new MyWork1();
		Test007.MyWork2 work2 = test007.new MyWork2();
		Test007.MyWork3 work3 = test007.new MyWork3();
		work1.start();
		work2.start();
		work3.start();
	}
}
```

8. Test008.java

Deadlock occurs in Test008.java. Test008.java has 3 Threads. We have obj1, obj2 and obj3 that are created from the Object Class. Thread1 locks obj1 and obj2 sequentially, Thread2 locks obj2 and obj3 sequentially and Thread3 locks obj3 and obj1 sequentially. The lock orders of three threads are cyclic that cause deadlock occur. The conditions of Test008.java are the Parallel Condition, the Reachable Condition, the Escaping Condition, the Non-Guarded Lock Condition and the Cyclic Lock Dependency Condition. The source code of Test008.java is shown in Table 27.

Table 27 The source code of Test008.java that deadlock occurs

```
package test.condition.Escaping;

public class Test008 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj2";
	private Object obj3 = "obj3";

	class MyWork1 extends Thread {
		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				synchronized (obj2) {
					System.out.println("obj2 is locked");
				}
			}
			System.out.println("finish work1");
		}
	}

	class MyWork2 extends Thread {
		public void run() {
			synchronized (obj2) {
				System.out.println("obj2 is locked.");
				synchronized (obj3) {
					System.out.println("obj3 is locked.");
				}

			}
			System.out.println("finish work2");
		}
	}
```

```
        class MyWork3 extends Thread {
                public void run() {
                        synchronized (obj3) {
                                System.out.println("obj3 is locked.");
                                synchronized (obj1) {
                                        System.out.println("obj1 is locked.");
                                }
                        }
                        System.out.println("finish work3");
                }
        }

        public static void main(String[] args) {
                Test008 test008 = new Test008();
                Test008.MyWork1 work1 = test008.new MyWork1();
                Test008.MyWork2 work2 = test008.new MyWork2();
                Test008.MyWork3 work3 = test008.new MyWork3();
                work1.start();
                work2.start();
                work3.start();
        }
}
```

9. Test009.java

In Test009.java occurs deadlock. Test009.java has 2 Threads. We have obj1, obj2 and obj4 that are created from the Object Class. Thread1 locks obj1 and obj2 sequentially and Thread2 locks obj4, obj2 and obj1 sequentially. Deadlock occurs because the order of obj1 and obj2 from both threads are reverse order. The conditions of Test009.java are the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Lock Dependency Condition. The source code of Test009.java is shown in Table 28.

Table 28 The source code of Test009.java that deadlock occurs

```java
package test.condition.nonGuarded;

public class Test009 {

        Object obj1 = "obj1";
        Object obj2 = "obj2";
        Object obj4 = "obj4";

        class MyWork1 extends Thread {
                public void run() {
                        synchronized (obj1) {
                                System.out.println("obj1 is locked.");
                                synchronized (obj2) {
                                        System.out.println("obj2 is locked.");
                                }
                        }
                        System.out.println("finish work1.");
                }
        }
        class MyWork2 extends Thread {
                public void run() {
                        synchronized (obj4) {
                                System.out.println("obj4 is locked - perform 
guarded lock.");
                                synchronized (obj2) {
                                        System.out.println("obj2 is locked.");
                                        synchronized (obj1) {
                                                System.out.println("obj1 is 
locked.");
                                        }
                                }
                        }
                        System.out.println("finish work2.");
                }
        }
```

```
	public static void main(String[] args) {
		Test009 test009 = new Test009();
		Test009.MyWork1 work1 = test009.new MyWork1();
		Test009.MyWork2 work2 = test009.new MyWork2();
		work1.start();
		work2.start();
	}
}
```

10. Test010.java

Deadlock in Test010.java does not occur. Test010.java has 2 Threads. We have obj1, obj2 and obj4 that are created from the Object Class. Thread1 locks obj4, obj1 and obj2 sequentially and Thread2 locks obj4, obj2 and obj1 sequentially. However, obj1 and obj2 from both threads are reverse orders, it does not occur deadlock because both threads has a guarded lock that is obj4. The conditions of Test010.java are the Parallel Condition, the Escaping Condition and the Reachable Condition. The source code of Test010.java is shown in Table 29.

Table 29 The source code of Test010.java that deadlock does not occur

```
package test.condition.nonGuarded;

public class Test010 {

	Object obj1 = "obj1";
	Object obj2 = "obj2";
	Object obj4 = "obj4";

	class MyWork1 extends Thread {

		public void run() {
			synchronized (obj4) {
				System.out.println("obj4 is locked - 
perform guarded lock.");
				synchronized (obj1) {
					System.out.println("obj1 is 

locked.");
					synchronized (obj2) {
						System.out.println("obj2 is 

locked.");
					}
				}
			}
			System.out.println("finish work1.");
		}
	}
	class MyWork2 extends Thread {
		public void run() {
			synchronized (obj4) {
				System.out.println("obj4 is locked - 
perform guarded lock.");
				synchronized (obj2) {
					System.out.println("obj2 is 
locked.");
					synchronized (obj1) {
						System.out.println("obj1 is 
locked.");
					}
				}
			}
			System.out.println("finish work2.");
		}
	}

	public static void main(String[] args) {
		Test010 test010 = new Test010();
		Test010.MyWork1 work1 = test010.new MyWork1();
		Test010.MyWork2 work2 = test010.new MyWork2();
		work1.start();
		work2.start();
	}

}
```

11. Test011.java

Deadlock occurs in Test011.java. Test011.java has 2 Threads. We have obj1, obj2 and obj3 that are created from the Object Class. Thread1 locks obj1, obj3 and obj2 sequentially and Thread2 locks obj2 and obj1 sequentially. Deadlock occurs from reverse order of obj1 and obj2 from both threads. The conditions of Test011.java are the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Lock Dependency Condition. The source code of Test011.java is shown in Table 30.

Table 30 The source code of Test011.java that deadlock occurs

```
package test.condition.Reachable;

public class Test011 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj2";
	private Object obj3 = "obj3";

	public class MyWork1 extends Thread {
		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				synchronized (obj3) {
					System.out.println("obj3 is locked.");
					synchronized (obj2) {
						System.out.println("obj2 is 
locked.");
					}
				}
			}
			System.out.println("finish work1.");
		}
	}


	public class MyWork2 extends Thread {
		public void run() {
			synchronized (obj2) {
				System.out.println("obj1 is locked.");
				synchronized (obj1) {
					System.out.println("obj2 is locked.");
				}
			}
			System.out.println("finish work2");
		}
	}
	public static void main(String[] args) {
		Test011 test011 = new Test011();
		Test011.MyWork1 work1 = test011.new MyWork1();
		Test011.MyWork2 work2 = test011.new MyWork2();
		work1.start();
		work2.start();
	}
}
```

12. Test012.java

Deadlock does not occur in Test012.java. Test012.java has 2 Threads. We have obj1, obj2 and obj3 that are created from the Object Class. Thread1 locks obj1 and obj2 sequentially and Thread2 locks obj1, obj3 and obj2 sequentially. Deadlock does not occur because lock orders of both threads are not reverse order. The conditions of Test012.java are the Parallel Condition, the Escaping Condition and the Reachable Condition. The source code of Test012.java is shown in Table 31.

Table 31 The source code of Test012.java that deadlock does not occur

```
package test.condition.Reachable;

public class Test012 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj2";
	private Object obj3 = "obj3";

	class MyWork1 extends Thread {

		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				synchronized (obj2) {
					System.out.println("obj2 is locked");
				}
			}
			System.out.println("finish work1.");
		}
	}

	class MyWork2 extends Thread {
		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				synchronized (obj3) {
					System.out.println("obj3 is locked.");
					synchronized (obj2) {
						System.out.println("obj2 is
locked.");
```

```
                                        }
                                }
                        }
                        System.out.println("finish work2.");
                }
        }

        public static void main(String[] args) {
                Test012 test012 = new Test012();
                Test012.MyWork1 work1 = test012.new MyWork1();
                Test012.MyWork2 work2 = test012.new MyWork2();
                work1.start();
                work2.start();
        }
}
```

13. Test013.java

Deadlock occurs in Test013.java. Test013.java has 2 Threads. We have obj1, obj2 and obj3 that are created from the Object Class. Thread1 locks obj1, obj2 and obj3 sequentially and Thread2 locks obj3, obj1 and obj2 sequentially. Deadlock occurs because the lock orders of obj1 and obj3 of both threads are reverse orders. The conditions of Test013 that are the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Lock Dependency Condition. The source code of Test013.java is shown in Table 32.

Table 32 The source code of Test013.java that deadlock occurs

```
package test.condition.Reachable;

public class Test013 {

        private Object obj1 = "obj1";
        private Object obj2 = "obj2";
        private Object obj3 = "obj3";

        class MyWork1 extends Thread {

                public void run() {
                        synchronized (obj1) {
                                System.out.println("obj1 is locked.");
                                synchronized (obj2) {
                                        System.out.println("obj2 is locked");
                                        synchronized (obj3) {
                                                System.out.println("obj3 is 
locked.");
                                        }
                                }
                        }
                        System.out.println("finish work1.");
                }
        }


        class MyWork2 extends Thread {
                public void run() {
                        synchronized (obj3) {
                                System.out.println("obj3 is locked.");
                                synchronized (obj1) {
                                        System.out.println("obj1 is locked.");
                                        synchronized (obj2) {
                                                System.out.println("obj2 is
locked.");
                                        }
                                }
                        }
                        System.out.println("finish work2.");
                }
        }

        public static void main(String[] args) {
                Test013 test013 = new Test013();
                Test013.MyWork1 work1 = test013.new MyWork1();
                Test013.MyWork2 work2 = test013.new MyWork2();
                work1.start();
                work2.start();
        }
}
```

14. Test014.java

Deadlock occurs in Test014.java. Test014.java has 2 Threads. We have obj1, obj2 and obj3 that are created from the Object Class. Thread1 locks obj1, obj2 and obj3sequentially and Thread2 locks obj2, obj1 and obj3sequentially. Deadlock occurs because the lock orders of obj1 and obj2 of both threads are reverse. The conditions of Test014.java are the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Lock Dependency Condition. The source code of Test014.java is shown in Table 33.

Table 33 The source code of Test014.java that deadlock occurs

```
package test.condition.Reachable;

public class Test014 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj2";
	private Object obj3 = "obj3";

	class MyWork1 extends Thread {

		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				synchronized (obj2) {
					System.out.println("obj2 is locked");
					synchronized (obj3) {
						System.out.println("obj3 is 
locked.");
					}
				}
			}
			System.out.println("finish work1.");
		}
	}

	class MyWork2 extends Thread {
		public void run() {
			synchronized (obj2) {
				System.out.println("obj2 is locked.");
				synchronized (obj1) {
					System.out.println("obj1 is locked.");
					synchronized (obj3) {
						System.out.println("obj3 is 
locked.");
					}
				}
			}
			System.out.println("finish work2.");
		}
	}

	public static void main(String[] args) {
		Test014 test013 = new Test014();
		Test014.MyWork1 work1 = test013.new MyWork1();
		Test014.MyWork2 work2 = test013.new MyWork2();
		work1.start();
		work2.start();
	}
}
```

15. Test015.java

Deadlock occurs in Test015.java. Test015.java has 2 Threads. Test015.java extends Thread and implements Runnable. We have obj1 and obj2. Thread1 locks obj1 and obj2 sequentially and Thread2 locks obj2 and obj1 sequentially. Deadlock occurs because the lock orders of obj1 and obj2 of both threads are reverse. The conditions of Test015.java are the Parallel Condition, the Escaping Conditions, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Lock Dependency Condition. The source code of Test015.java is shown in Table 34.

Table 34 The source code of Test015.java that deadlock occurs

```
package test.condition.ExRunnable;

public class Test015 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj2";

	public class MyWork1 extends Thread implements Runnable{
		public void run(){
			synchronized (obj1) {
				System.out.println("obj1 is locked");
				synchronized (obj2) {
					System.out.println("obj2 is locked");
				}
			}
			System.out.println("finish work1.");
		}
	}

	public class MyWork2 extends Thread implements Runnable{
		public void run(){
			synchronized(obj2){
				System.out.println("obj2 is locked.");
				synchronized (obj1) {
					System.out.println("obj1 is locked.");
				}
			}
```

```
                        System.out.println("finish work2.");
                }
        }

        public static void main(String[] args) {
                Test015 test015 = new Test015();
                Test015.MyWork1 work1 = test015.new MyWork1();
                Test015.MyWork2 work2 = test015.new MyWork2();
                work1.start();
                work2.start();
        }
}
```

16. Test016.java

Deadlock does not occur in Test016.java. Test016.java has 2 Threads. Test016.java extends Thread and implements Runnable. We have obj1 and obj2. Thread1 locks obj1 and obj2 sequentially and Thread2 locks obj1 and obj2 sequentially. Deadlock does not occur because the lock orders of obj1 and obj2 of both threads are not reverse. The conditions of Test016.java are the Parallel Condition, the Escaping Condition and the Reachable Condition. The source code of Test016.java is shown in Table 35.

Table 35 The source code of Test016.java that deadlock does not occur

```
package test.condition.ExRunnable;

public class Test016 {

        private Object obj1 = "obj1";
        private Object obj2 = "obj2";

        public class MyWork1 extends Thread implements Runnable {
                public void run() {
                        synchronized (obj1) {
                                System.out.println("obj1 is locked");
                                synchronized (obj2) {
                                        System.out.println("obj2 is locked");
                                }
                        }
                        System.out.println("finish work1.");
                }
        }

        public class MyWork2 extends Thread implements Runnable {
                public void run() {
                        synchronized (obj1) {
                                System.out.println("obj2 is locked.");
                                synchronized (obj2) {
                                        System.out.println("obj1 is
locked.");
                                        }
                              }
                              System.out.println("finish work2.");
                    }
          }

          public static void main(String[] args) {
                    Test016 test015 = new Test016();
                    Test016.MyWork1 work1 = test015.new MyWork1();
                    Test016.MyWork2 work2 = test015.new MyWork2();
                    work1.start();
                    work2.start();
          }
}
```

17. Test017.java

Deadlock occurs in Test017.java. Test017.java has 3 Threads. Test017.java extends Thread and implements Runnable. We have obj1, obj2 and obj3. Thread1 locks obj1 and obj2 sequentially, Thread2 locks obj2 and obj3 sequentially and Thread3 locks obj3 and obj1 sequentially. Deadlock occurs because the lock orders of obj1, obj2 and obj3 of three threads are reverse order. The conditions of Test017.java are the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Lock Dependency Condition. The source code of Test017.java is shown in Table 36.

Table 36 The source code of Test017.java that deadlock occurs

```
package test.condition.ExRunnable;

public class Test017 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj2";
	private Object obj3 = "obj3";

	public class MyWork1 extends Thread implements Runnable {
		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked");
				synchronized (obj2) {
					System.out.println("obj2 is locked");
				}
			}
			System.out.println("finish work1.");
		}
	}

	public class MyWork2 extends Thread implements Runnable {
		public void run() {
			synchronized (obj2) {
				System.out.println("obj2 is locked.");
				synchronized (obj3) {
					System.out.println("obj3 is locked.");
				}
			}
			System.out.println("finish work2.");
		}
	}

	public class MyWork3 extends Thread implements Runnable {
		public void run() {
			synchronized (obj3) {
				System.out.println("obj3 is locked.");
				synchronized (obj1) {
```

```
                                        System.out.println("obj1 is locked.");
                                }
                        }
                        System.out.println("finish work3.");
                }
        }

        public static void main(String[] args) {
                Test017 test015 = new Test017();
                Test017.MyWork1 work1 = test015.new MyWork1();
                Test017.MyWork2 work2 = test015.new MyWork2();
                Test017.MyWork3 work3 = test015.new MyWork3();
                work1.start();
                work2.start();
                work3.start();
        }
}
```

18. Test018.java

Deadlock occurs in Test018.java. Test018.java has 2 Threads. Test018.java extends Thread and implements Runnable. We have obj1, obj2 and obj3 that are created from the Object Class. Thread1 locks obj1, obj2 and obj3 sequentially and Thread2 locks obj2, obj3 and obj1 sequentially. Deadlock occurs because lock orders of obj1, obj2 and obj3 of both threads are reverse. The conditions of Test018.java are the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Lock Dependency Condition. The source code of Test018.java is shown in Table 37.

Table 37 The source code of Test018.java that deadlock occurs

```
package test.condition.ExRunnable;

public class Test018 {

	private Object obj1 = "obj1";
	private Object obj2 = "obj2";
	private Object obj3 = "obj3";

	public class MyWork1 extends Thread implements Runnable {
		public void run() {
			synchronized (obj1) {
				System.out.println("obj1 is locked");
				synchronized (obj2) {
					System.out.println("obj2 is locked");
					synchronized (obj3) {
						System.out.println("obj3 is 
locked");
					}
				}
			}
			System.out.println("finish work1.");
		}
	}

	public class MyWork2 extends Thread implements Runnable {
		public void run() {
			synchronized (obj2) {
				System.out.println("obj2 is locked.");
				synchronized (obj3) {
					System.out.println("obj3 is locked.");
					synchronized (obj1) {
						System.out.println("obj1 is 
locked");
					}
				}
```

```
                    }
                    System.out.println("finish work2.");
            }
    }

    public static void main(String[] args) {
            Test018 test015 = new Test018();
            Test018.MyWork1 work1 = test015.new MyWork1();
            Test018.MyWork2 work2 = test015.new MyWork2();
            work1.start();
            work2.start();
    }
}
```

19. Test019.java

Deadlock does not occur in Test019.java. Test019.java has 3 Threads. We have obj1, obj2, obj3 and obj4 that are created from the Object Class. Thread1 locks obj1 and obj2 sequentially, Thread2 locks obj1 and obj3 sequentially and Thread3 locks obj1 and obj4 sequentially. Deadlock does not occur because obj1 is a guarded lock for all threads. The conditions of Test019.java are the Parallel Condition, the Escaping Condition and the Reachable Condition. The source code of Test019.java is shown in Table 38.

Table 38 The source code of Test019.java that deadlock does not occur

```
package test.condition.Parallel;

public class Test019 {

    private Object obj1 = "obj1";
    private Object obj2 = "obj2";
    private Object obj3 = "obj3";
    private Object obj4 = "obj4";


    class MyWork1 extends Thread {
            public void run() {
                    synchronized (obj1) {
                            System.out.println("obj1 is locked.");
                            synchronized (obj2) {
                                    System.out.println("obj2 is locked");
                            }
                    }
                    System.out.println("finish work1");
            }
    }
```

```
        class MyWork2 extends Thread {
                public void run() {
                        synchronized (obj1) {
                                System.out.println("obj1 is locked.");
                                synchronized (obj3) {
                                        System.out.println("obj3 is locked.");
                                }

                        }
                        System.out.println("finish work2");
                }
        }

        class MyWork3 extends Thread {
                public void run() {
                        synchronized (obj1) {
                                System.out.println("obj1 is locked.");
                                synchronized (obj4) {
                                        System.out.println("obj4 is locked.");
                                }
                        }
                        System.out.println("finish work3");
                }
        }

        public static void main(String[] args) {
                Test019 test019 = new Test019();
                Test019.MyWork1 work1 = test019.new MyWork1();
                Test019.MyWork2 work2 = test019.new MyWork2();
                Test019.MyWork3 work3 = test019.new MyWork3();
                work1.start();
                work2.start();
                work3.start();
        }
}
```

20. Test020.java

Deadlock does not occur in Test020.java. Test020.java has no Thread. The source code of Test020.java does not relate to deadlock. It does not implement Thread or Runnable, although it has synchronized block and calls the wait() method and the notify() method. There is no condition reported for Test020.java. The source code of Test020.java is shown in Table 39.

Table 39 The source code of Test020.java that deadlock does not occur

```
package test.condition.Misc;

public class Test020 {

    private boolean pizzaArrived = false;

    public void eatPizza() throws InterruptedException{
        synchronized(this){
            while(!pizzaArrived){
             wait();
            }
        }
        System.out.println("yumyum..");
    }
    public void pizzaGuy(){
        synchronized(this){
             this.pizzaArrived = true;
             notifyAll();
        }
    }
}
```

21. Test021.java

In Test021.java deadlock does not occur because there is only one thread is created in Test021.java and the thread locks only one object and call the wait() method to wait until the task is done and then it calls the notify() method. Test021.java can be detected the Non-Guarded Lock Condition. The source code of Test021.java is shown in Table 40.

Table 40 The source code of Test021.java that deadlock does not occur

```java
package test.condition.WaitNotify;

public class Test021 {

	class MyWork1 extends Thread {
		int total;
		public void run() {
			synchronized (this) {
				for (int i = 0; i < 100; i++) {
					total += 1;
				}
				notify();
			}
		}
	}

	public static void main(String[] args) {
		Test021 test021 = new Test021();
		Test021.MyWork1 work1 = test021.new MyWork1();
		work1.start();
		synchronized (work1) {
			try {
				System.out.println("waiting work1.");
				work1.wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println(work1.total);
		}
	}
}
```

22. Test022.java

In Test022.java deadlock does not occur because Test022.java is implemented to lock ‘this’ object that in Java, ‘this’ object is locked from the start point of the application and there is only one object that is locked therefore deadlock does not occur, although there are 2 Threads; Thread1 and Thread2, created. The conditions of Test022.java are the Aliasing Condition, the Parallel Condition, the Escaping Condition, the Reachable Condition and the Non-Guarded Lock Condition. The source code of Test022.java is shown in Table 41.

Table 41 The source code of Test022.java that deadlock does not occur

```
package test.condition.WaitNotify;

public class Test022 {

	int total = 0;
	Object obj1 = "obj1";
	Object obj2 = "obj2";

	class MyWork1 extends Thread {
		public void run() {
			synchronized (this) {
				System.out.println("this thread is locked.");
				for (int i = 0; i < 100; i++) {
					total += 1;
				}
				notify();
			}
			System.out.println("finish work1");
		}
	}

	class MyWork2 extends Thread {

		public void run() {
			synchronized (this) {
				System.out.println("this thread is locked.");
				for (int i = 0; i < 100; i++) {
					total += 1;
				}
				notify();
			}
			System.out.println("finish work2");
		}
	}
	public static void main(String[] args) {
		Test022 test022 = new Test022();
		Test022.MyWork1 work1 = test022.new MyWork1();
		Test022.MyWork1 work2 = test022.new MyWork1();
		work1.start();
		work2.start();
		synchronized (work1) {
			try {
				System.out.println("waiting for work1.");
				work1.wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			synchronized (work2) {
				try {
					System.out.println("waiting for 
work2");
					work2.wait();
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		}
		work1.notify();
		work2.notify();
		System.out.println("tasks are complete.");
	}
}
```

23. Test023.java

In Test023.java deadlock does not occur because Test023.java has 2 threads; Thread1 and Thread2, lock the same object that is obj1. The deadlock conditions of Test023.java are the Aliasing Condition, the Parallel Condition, Escaping Condition and Non-Guarded Lock Condition. The source code of Test023.java is shown in Table 42.

Table 42 The source code of Test023.java that deadlock does not occur

```
package test.condition.WaitNotify;
public class Test023 {
        Object obj1 = "obj1";
        class MyWork1 extends Thread{
                public void run(){
                        doJob();
                }
                public void doJob(){
                        synchronized(obj1){
                                System.out.println("obj1 is locked.");
                                try {
                                        System.out.println("obj1 is in wait 
state...");
                                        obj1.wait();
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                        System.out.println("finish job work1.");
                }
        }
        class MyWork2 extends Thread{
                public void run(){
                        trickerRelease();
                }
                public void trickerRelease(){
                        System.out.println("obj1 is released.");
                        synchronized (obj1) {
                                obj1.notify();
                        }
                }
        }
        public static void main(String[] args) {
                Test023 test023 = new Test023();
                Test023.MyWork1 work1 = test023.new MyWork1();
                Test023.MyWork2 work2 = test023.new MyWork2();
                work1.start();
                work2.start();
                System.out.println("done.");
        }
}
```

24. Test024.java

In Test024.java deadlock occurs. Test024.java has 2 Threads. We have obj1 and obj2 that are created from the Object Class. Thread1 locks obj1 and obj2 sequentially and Thread 2 locks obj2 and obj1 sequentially. The lock order of both threads is reverse. The conditions of Test024.java are the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Lock Dependency Condition. The source code of Test024.java is shown in Table 43.

Table 43 The source code of Test024.java that deadlock does not occur

```
package test.condition.WaitNotify;

public class Test024 {

        Object obj1 = "obj1";
        Object obj2 = "obj2";

        class MyWork1 extends Thread {
                public void run() {
                        doJob();
                }

                public void doJob() {
                        synchronized (obj1) {
                                System.out.println("obj1 is locked.");
                                try {
                                        System.out.println("obj1 is in wait
state...");
                                        obj1.wait();
                                        synchronized (obj2) {
                                                System.out.println("obj2 is in
wait state...");
                                                obj2.wait();
                                        }
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                        System.out.println("finish job work1.");
                }
        }
```

```
	class MyWork2 extends Thread {
		public void run() {
			trickerRelease();
		}

		public void trickerRelease() {
			System.out.println("obj1 is released.");
			synchronized (obj2) {
					obj2.notify();
				synchronized (obj1) {
						obj1.notify();
				}
			}
		}
	}

	public static void main(String[] args) {
		Test024 ex3 = new Test024();
		Test024.MyWork1 work1 = ex3.new MyWork1();
		Test024.MyWork2 work2 = ex3.new MyWork2();
		work1.start();
		work2.start();
		System.out.println("done.");
	}
}
```

25. Test025.java

In Test025.java deadlock occurs. Test025.java has 4 Threads. We have obj1 and obj2 that are created from the Object Class. Thread 1 locks obj1, calls obj1.wait(), locks obj2 and calls obj2.wait() sequentially and Thread2 locks obj2, calls obj2.notify(), locks obj1 and calls obj1.notify() sequentially. We create Thread3 that has the same lock sequence as Thread1 and Thread4 that has the same lock sequence as Thread2. When there are more than 2 threads, it is possible that deadlock occurs because of race condition of Thread executing. If Thread2 or Thread4 starts first and calls the notify() method for both object, another thread; Thread1 or Thread3 and the rest of thread can lock objects in reverse order and cause deadlock. The conditions of Test025.java are the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Dependency Condition. The source code of Test025.java is shown in Table 44.

Table 44 The source code of Test025.java that deadlock occurs

```
package test.condition.WaitNotify;

public class Test025 {

	Object obj1 = "obj1";
	Object obj2 = "obj2";

	class MyWork1 extends Thread {
		public void run() {
			doJob();
		}

		public void doJob() {
			synchronized (obj1) {
				System.out.println("obj1 is locked.");
				try {
					System.out.println("obj1 is in wait 
state...");
					obj1.wait();
					synchronized (obj2) {
						System.out.println("obj2 is in 
wait state...");
						obj2.wait();
					}
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
```

```
                        }
                        System.out.println("finish job work1.");
                }
        }

        class MyWork2 extends Thread {
                public void run() {
                        trickerRelease();
                }

                public void trickerRelease() {
                        System.out.println("obj1 is released.");
                        synchronized (obj2) {
                                obj2.notify();
                                synchronized (obj1) {
                                        obj1.notify();
                                }
                        }
                }
        }

        public static void main(String[] args) {
                Test025 ex3 = new Test025();
                Test025.MyWork1 work11 = ex3.new MyWork1();
                Test025.MyWork1 work12 = ex3.new MyWork1();
                Test025.MyWork2 work21 = ex3.new MyWork2();
                Test025.MyWork2 work22 = ex3.new MyWork2();
                work11.start();
                work12.start();
                work21.start();
                work22.start();
                System.out.println("done.");
        }
}
```

26. Test026.java

In Test026.java deadlock occurs. Test026.java has 2 Threads. We have the ‘from’ Object and the ‘to’ Object that are created from the Test026 Class. Thread1 locks the ‘from’ Object and the ‘to’ Object sequentially and Thread2 locks the ‘to’ Object and the ‘from’ Object sequentially. The lock orders of both Threads are reverse order therefore deadlock occurs. The conditions of Test026.java are the Parallel Condition, the Escaping Condition, the Reachable Condition, the Non-Guarded Lock Condition and the Cyclic Dependency Condition. The source code of Test026.java is shown in Table 45.

Table 45 The source code of Test026.java that deadlock occurs

```java
package test.condition.Test;

public class Test026 {

	double balance;

	public Test026(double balInit) {

		this.balance = balInit;

	}

	public void debit(double val) {

		balance += val;

	}

	public void credit(double val) {

		balance -= val;

	}

	public double getBalance() {

		return balance;

	}
	static class Test026_2 {
		public void pay(Test026 from, Test026 to, double val) {
			System.out.println("balance transfer...");
			System.out.println("lock acquired to...");
			synchronized (to) {
				System.out.println("lock acquired from...");
				synchronized (from) {
					if (from.getBalance() >= val) {
						from.debit(val);
						to.credit(val);
						System.out.println("pay
finished...");
					}
				}
			}
		}
	}
	static class Test026_1 {
		public void transfer(Test026 from, Test026 to, double val) {
			System.out.println("balance transfer...");
			System.out.println("lock acquired from...");
			synchronized (from) {
				System.out.println("lock acquired to...");
				synchronized (to) {
					if (from.getBalance() >= val ) {
						from.debit(val);
						to.credit(val);
						System.out.println("transfer
finished...");
```

```
		public void deposit(Test026 to, double val) {

			System.out.println("money deposit");

			System.out.println("lock acquired to...");

			synchronized (to) {

				to.credit(val);

				System.out.println("deposit finished...");

			}

		}
	}

	public static void main(String[] args) {

		final Test026 cashier1 = new Test026(60000);

		final Test026 cashier2 = new Test026(80000);

		final Test026_1 opc = new Test026_1();
		final Test026_2 opc2 = new Test026_2();

		new Thread(new Runnable() {

			public void run() {

				opc.transfer(cashier1, cashier2, 20000);

			}

		}).start();

		new Thread(new Runnable() {

			public void run() {

				opc2.pay(cashier1, cashier2, 20000);

			}

		}).start();

	}
}
```

VITA

Name	Suvarin Ploysri
Gender	Female
Date of Birth	February 14, 1984
Place of Birth	Chiang Mai, Thailand
Education	
2014	M.Sc. in Software Engineering, Chulalongkorn University
2005	B.Eng. in Computer Engineering, Chiang Mai University