Automated Deadlock Detection for Large Java Libraries

R. Rajesh Kumar(✉), Vivek Shanbhag, and K. V. Dinesha

International Institute of Information Technology,
Bangalore 560100, Karnataka, India
rajesh.kumar@iiitb.org
https://www.iiitb.ac.in

Abstract. Locating deadlock opportunities in large Java libraries is a subject of much research as the Java Execution Environment (JVM /JRE) does not provide means to predict or prevent deadlocks. Researchers have used static and dynamic approaches to analyze the problem.

Static approaches: With very large libraries, this analysis face typical accuracy/doability problem. If they employ a detailed modelling of the library, then the size of the analysis grows too large. Instead, if their model is coarse grained, then the results have too many false cases. Since they do not generate deadlocking test cases, manually creating deadlocking code based on the predictions is impractical for large libraries.

Dynamic approaches: Such analysis produces concrete results in the form of actual test cases to demonstrate the reachability of the identified deadlock. Unfortunately, for large libraries, generating the seed test execution paths covering all possible classes, to trigger the dynamic analysis becomes impractical.

In this work we combine a static approach (Stalemate) and a dynamic approach (Omen) to detect deadlocks in large Java libraries. We first run 'Stalemate' to generate a list of potential deadlocking classes. We feed this as input test case to Omen. In case of deadlock, details are logged for subsequent reproduction. This process is automated without the need for manual intervention.

We subjected the entire JRE v1.7.0_79 libraries (rt.jar) to our implementation of the above approach and successfully detected 113 deadlocks. We reported a few of them to Oracle as defects. They were accepted as bugs.

Keywords: Concurrency · Deadlock · Java · Static analysis · Dynamic analysis · Scalable

1 Introduction

The *synchronised* construct is provided in the Java Language to facilitate the development of code fragments that may run concurrently. When this happens in an uncoordinated manner it can give rise to *Lock Order Violations*. If such a lock

© Springer Nature Switzerland AG 2021
D. Goswami and T. A. Hoang (Eds.): ICDCIT 2021, LNCS 12582, pp. 129–144, 2021.
https://doi.org/10.1007/978-3-030-65621-8_8

order violation is realised during program execution it causes the executing JVM to deadlock. This problem has been widely investigated, and various researchers have tried different approaches to address it.

There have been numerous published research works for detecting deadlocks including both static and dynamic approaches [1,2,4–6,9,11–16] and some for preventing them [3,7,8].

Stalemate [1,2] is a static analysis approach that identifies lock order violations in large Java libraries, which have the possibility of being realised, during execution, in the form of a deadlock. However, its predictions include many false positives, since the analysis is at the *type* level while the contested locks are held on the actual objects. Consequently, one must develop a deadlocking test case by manual examination of the call cycles in the predictions. For a library as large as the entire JRE, it becomes impractical to apply this manual method to narrow down to realisable deadlocks. Other static approaches such as Jade [12,13] can scale well, produces notable results, however these methods also produce many false positives.

There are many dynamic analysis approaches to identify deadlocks such as Omen [4], Needlepoint [5] and Sherlock [6]. Omen, produces realisable deadlocks along with reproducible test cases. Such dynamic analysis, initiated using seed test cases, is limited to the execution traces realisable during the call flows of the seed test cases. For large libraries, the set of seed test cases would become substantial, and looking for lock cycles realisable in their execution traces would be impractical. The implementation Sherlock is suitable for large programs but not for libraries.

In this paper we address the problem of identifying reproducible deadlocks in large Java libraries without false positives, and produce deadlocking test cases for the detected deadlocks. Our approach combines a static approach Stalemate [1,2], as it can scale to analyze large libraries and a dynamic approach Omen [4] to detect real deadlocks along with deadlocking test cases.

We start by subjecting the library to Stalemate and from the lock order violations it reports, interleaving calls that could lead to deadlocks are extracted. Reading off classes/methods involved in such calls, we use Randoop to generate test cases that exercise these classes/methods. These tests are then subjected to dynamic analysis (Omen) narrowing down to reproducible deadlocks. In this way we eliminate the numerous false cases identified by static analysis. At the same time the attention of the dynamic analysis is directed to only those components in the library where there is a possibility of finding a deadlock. We have fully automated the process so that the complete analysis for the entire library can be completed with no manual intervention.

This paper is organized as follows: Sect. 2 states the problem we are attempting to address. Section 3 provides the solution overview and details the implementation, Sect. 4 summarizes the results, and Sect. 5 states the conclusions.

2 Problem Statement

Design an automated method to analyze large Java libraries to detect deadlocks by combining static and dynamic analysis approaches. For each of the lock order violations identified by the static analysis (Stalemate), create an automated process to detect any real deadlock associated with that violation using dynamic analysis (Omen). The process should also generate the deadlocking test cases to reproduce the deadlocks.

As an illustration, we have used the output of Stalemate [1] on the entire JRE v1.7.0_79 libraries (rt.jar) to generate a list of lock order violations (this number is more than 26,000) with many false positive cases. We have used the automated process described in this paper to generate a list of real deadlocks (number: 113) and test cases to reproduce the deadlocks.

3 Solution Details

The solution focuses on extracting the relevant details from the static analysis and targets the dynamic analysis only on those classes that have the potential to cause a deadlock. Given a lock order violation that is identified by the static analysis *Stalemate*, next, we want to develop single-threaded test programs that may individually realise each its thread stacks. For this purpose we use the *Randoop* tool, which synthesizes test cases for a given set of classes. The dynamic analysis tool *Omen* is then invoked by passing the generated test case as the seed test case. On completion of the analysis if deadlocks are found, Omen will synthesize a multi-threaded test case that will always deadlock.

The task is to use these tools, namely, *Stalemate*, *Randoop* and *Omen*, and devise an automated method to analyze large Java libraries to detect deadlocks and produce deadlocking test scenarios. The flow is designed to handle any exceptions and intermittent failures so that it is suitable to analyze large libraries and results in a truly automated solution.

3.1 Solution Steps

The process flow is represented in Fig. 1. Key steps of the automated analysis is described below.

1. **Static Analysis:** The jar file containing the libraries that are to be analyzed are subjected to the static analysis. This step produces the static analysis results (Sn), which contains the list of potential deadlocking scenarios containing the call flow details with the lock order violations involved in the synchronized functions calls, as represented in Fig. 2.

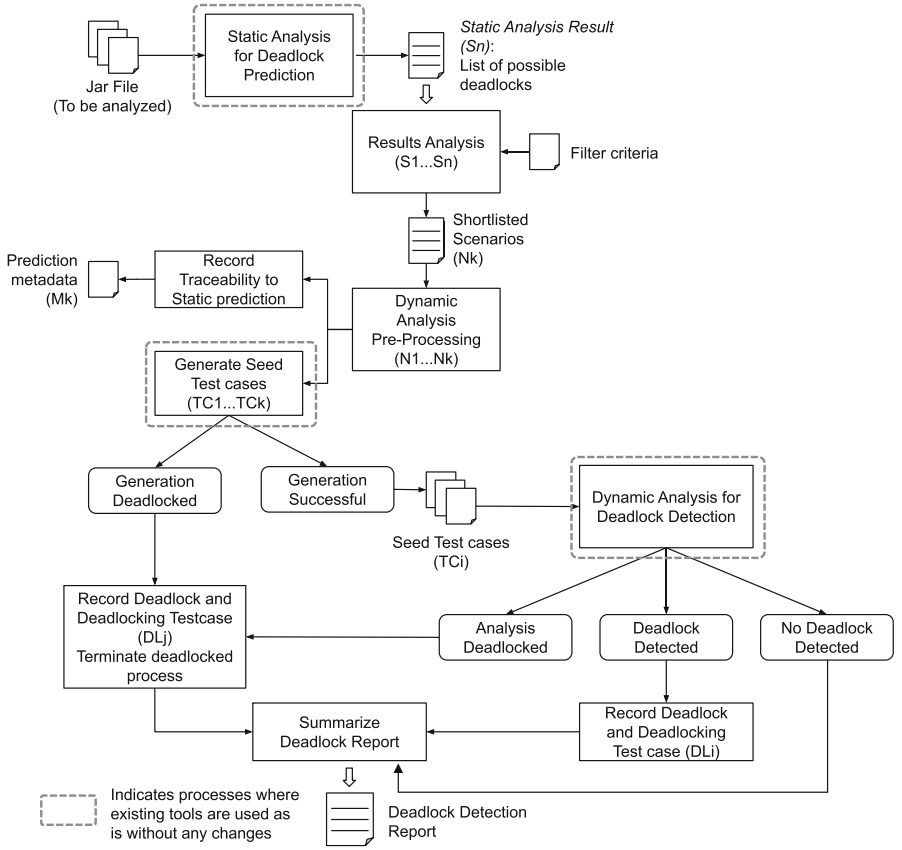

Fig. 1. Automated deadlock detection process flow

The result is interpreted as follows: the "Cycle-2" in its opening line means that is a lock order violation involving 2 locks. The class names of the locks are in the string that follows. This line is followed by a number of thread stacks enclosed within "Thread-i Option" descriptors. i ranging from 1 to n, the number of locks involved in the violation.

The example in Fig. 2, where $n = 2$, we refer to the "Thread-1" stacks as forward stacks, which acquire locks in the *forward order*, and the "Thread-2" stacks as reverse stacks, as they acquire them in the *reverse order*. If in a multi-threaded program, one thread were to realise one of the forward stacks, and another thread were to realise one of the reverse stacks, and if they were to do this concurrently, and if the two locks they are trying to acquire were to be the *same* two locks, then they could result in a deadlock.

```
<Cycle-2 java.util.logging.Logger.class java.util.logging.LogManager>

    <Thread-1 Option>
java.util.logging.Logger.getAnonymousLogger:()Ljava.util.logging.Logger;
java.util.logging.LogManager.getLogger:(Ljava.lang.String;)Ljava.util.logging.Logger;
    </Thread-1 Option>
<Thread-1 Option>
java.util.logging.Logger.getAnonymousLogger:(Ljava.lang.String;)Ljava.util.logging.Logger;
java.util.logging.LogManager.getLogger:(Ljava.lang.String;)Ljava.util.logging.Logger;
    </Thread-1 Option>
<Thread-1 Option>
java.util.logging.Logger.getLogger:(Ljava.lang.String;Ljava.lang.String;)Ljava.util.logging.Logger;
java.util.logging.LogManager.getLogger:(Ljava.lang.String;)Ljava.util.logging.Logger;
    </Thread-1 Option>
<Thread-1 Option>
java.util.logging.Logger.getLogger:(Ljava.lang.String;Ljava.lang.String;)Ljava.util.logging.Logger;
java.util.logging.LogManager.addLogger:(Ljava.util.logging.Logger;)Z
    </Thread-1 Option>
<Thread-1 Option>
java.util.logging.Logger.getLogger:(Ljava.lang.String;)Ljava.util.logging.Logger;
java.util.logging.LogManager.addLogger:(Ljava.util.logging.Logger;)Z
    </Thread-1 Option>
<Thread-1 Option>
java.util.logging.Logger.getLogger:(Ljava.lang.String;)Ljava.util.logging.Logger;
java.util.logging.LogManager.getLogger:(Ljava.lang.String;)Ljava.util.logging.Logger;
    </Thread-1 Option>

    <Thread-2 Option>
java.util.logging.LogManager.addLogger:(Ljava.util.logging.Logger;)Z
java.util.logging.Logger.getLogger:(Ljava.lang.String;)Ljava.util.logging.Logger;
    </Thread-2 Option>

</Cycle-2 java.util.logging.Logger.class java.util.logging.LogManager>
```

Fig. 2. Lock order violation output from static analysis

2. **Prediction Filtering and Data Preparation:** The results produced by Stalemate could be a very large data set. To narrow down the areas of focus, certain filters such as *Cycles*, *Call Depth* and *Call Density* are applied on the results (details of the filters are elaborated in Sect. 3.2). Each of the result nodes (S1 to Sn) are checked for the filter criteria and a subset Nk of the static analysis results ($Nk \subset Sn$) is produced. There is a possibility that this step could filter certain real deadlock candidates, however it helps to direct the analysis to the desired focus area. For each of the shortlisted nodes, the classes are extracted along with the metadata to trace back to the static analysis and it is referred as *Node Info*. An extract of a log created after the filtering exercise is shown in Fig. 3(a) and Node Info is shown in Fig. 3(b).
3. **Seed Test Case Generation:** Using the *Node Info* details collected during the data preparation step the seed test cases (TCk) are generated for each of the shortlisted predictions using the utility Randoop [10]. The test cases target specifically the classes involved in a lock order violation as detected by the static analysis. Example of a test case generated by Randoop is shown in Fig. 3(c).

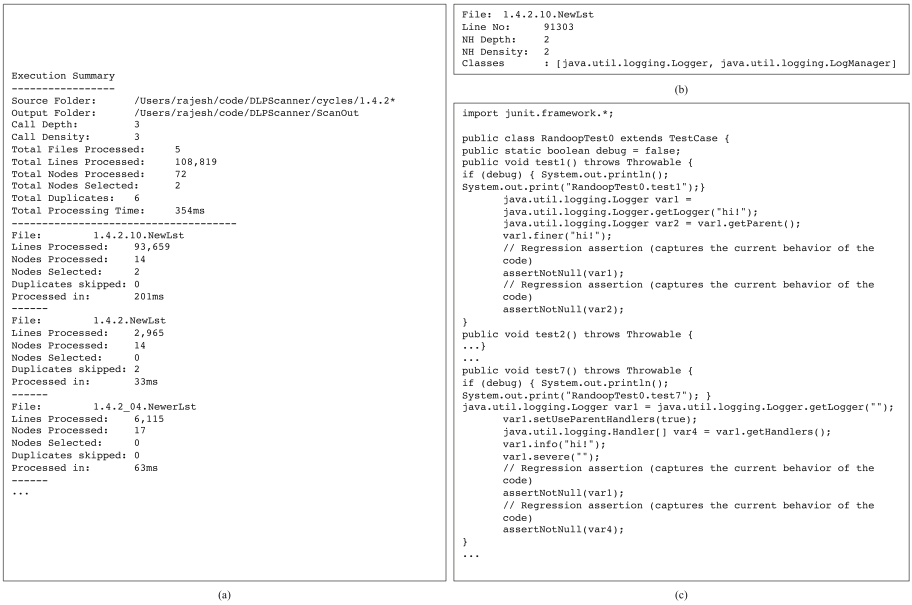

Fig. 3. (a) Execution summary (b) node info (c) test case generated by Randoop

4. **Dynamic Analysis for Deadlock Detection:** Post generation of the seed test cases, for each of the test cases (TC1 to TCk), the Dynamic Deadlock Analysis is initiated with Omen. The Dynamic Analysis starts by executing the seed test case and by tracing the lock dependency relations of the classes during the execution and recording them along with the invocation contexts. The presence of cyclic chains in the lock dependency relations are identified and potential deadlocking scenarios are synthesized. Multi-threaded tests are then generated by using the invocation details from the seed test case execution and by creating additional conditions that could result in a deadlock. At the end of the execution if any deadlocks are detected (DLj) successfully, they are recorded along with the deadlocking test cases. The class to be analyzed, seed test case, and the deadlocking test case generated by Omen are illustrated in Fig. 4 with a simple example. The flow continues by picking the next shortlisted result to be analyzed.
5. **Handling of Deadlocked Analysis:** It was observed that the execution of the analysis was getting deadlocked either during the generation of the seed test cases or during the dynamic analysis. It was observed that these deadlocking scenarios were occurring while processing specific set of classes. The stack traces of the deadlocked JVM revealed that these are indeed deadlocking scenarios as predicted by the static analysis and they were consistently occurring while processing those specific classes. The test case generation by Randoop is a multi-threaded process as it executes each test in a separate thread to speed up the generation. When the lock order violations occur in

the call flows of the classes for which the test cases are being generated, the possibility of deadlock occurs. Similarly, such scenarios occur during dynamic analysis as well. *JVM Monitoring Routine* was developed to watch the JVMs for such scenarios. Once a deadlocked JVM is detected then the stack traces were extracted and recorded along with the associated metadata so that it is traceable to the prediction. The hung JVM is then terminated by the routine so that the flow would continue to the next prediction to be analyzed. An extract of the JVM stack trace of such as scenario is shown in Fig. 5

```
public class Account {

int balance = 1000;

public void transfer(Account acc, int amount) {
   {
    synchronized(acc) { // To ensure atomicity
     if(acc != null && acc.debit(amount)) {
           credit(amount);
      }
    }
  }
}

public synchronized boolean debit(int amount) {
  if(balance < amount) return false;
  balance -= amount;
  return true;
}

public synchronized void credit(int amount) {
  balance += amount;
}

}
```

(a)

```
public class TestAccount {
  public static void main(String [] args) {
     Account user1, user2;
     user1 = new Account();
     user2 = new Account();
     // test credit
     user1.credit(200);
     // test debit
     user1.debit(100);
    // test transfer
    user1.transfer(user2,50);
  }
}
```

(b)

```
import java.util.*;
import java.io.*;
import omen.util.*;
public class TestDriver {

public static void cycle0() {
    Initializer.initialize(0);

     Thread thread0 = new Thread() {
     public void run() {
     List<Parameter> paramList =
           Initializer.collectedObjects.get(0);
     Account receiver =
            (Account)paramList.get(0).returnStored();
     Account par1 =
            (Account)paramList.get(1).returnStored();
     int par2 = (int)paramList.get(2).returnStoredInt();
     try {
        Thread.sleep(0);
        receiver.transfer(par1, par2);
     } catch (Exception e) {}
   }
};
Thread thread1 = new Thread() {
   public void run() {
   List<Parameter> paramList =
      Initializer.collectedObjects.get(1);
   Account receiver =
       (Account)paramList.get(0).returnStored();
   Account par1 =
       (Account)paramList.get(1).returnStored();
   int par2 =
        (int)paramList.get(2).returnStoredInt();
   try {
     Thread.sleep(1000);
     receiver.transfer(par1, par2);
   } catch (Exception e) {}
   }
};
//Start the threads
thread0.start();
thread1.start();
try {
   thread0.join();
   thread1.join();
} catch (Exception e) { }
}
/*
* Test all the cycles.
*/
public static void main(String args[]) {
       switch( Integer.parseInt(System.getenv()
                    .get("omen.errno"))) {
       case 0:
          cycle0();
          System.out.println("Printing cycle" +0);
          break;
       }
}
}
```

(c)

Fig. 4. Omen dynamic analysis example: (a) class to be analyzed (b) sequential seed test case to be subjected to the analysis (c) synthesized multi-threaded deadlocking test case generated by Omen

```
...

"Finalizer" daemon prio=5 tid=0x00007f8944813800 nid=0x3303 in
Object.wait() [0x0000700001e0e000]
   java.lang.Thread.State: WAITING (on object monitor)
       at java.lang.Object.wait(Native Method)
       - waiting on <0x0000000780004858> (a
java.lang.ref.ReferenceQueue$Lock)
       at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:135)
       - locked <0x0000000780004858> (a
java.lang.ref.ReferenceQueue$Lock)
       at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:151)
       at
java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)

...

"main" prio=5 tid=0x00007f8944802800 nid=0x1b03 runnable
[0x00007000016f8000]
   java.lang.Thread.State: RUNNABLE
       at
java.util.logging.Logger.getEffectiveResourceBundleName(Logger.java:1703
)
       at java.util.logging.Logger.doLog(Logger.java:636)
       at java.util.logging.Logger.log(Logger.java:664)
       at java.util.logging.Logger.info(Logger.java:1182)
       at RandoopTest0.test7(RandoopTest0.java:109)
       at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
...
```

Fig. 5. Extract of JVM stack trace

6. **Data Collection and Report Creation:** Comprehensive reports of the analysis are generated based on the data collected when the execution concludes, resulting in a comprehensive Deadlock Detection Report.

3.2 Implementation

The implementation details of the solution is elaborated in this section. The complete source code for of this implementation is published in GitHub along with the test results of the executions.

Execution Flow. The end-to-end execution flow design is represented in the Fig. 6. The flow of the automated deadlock analyzer is as follows (refer Fig. 6, steps 1 to 7):

1. The *run cycles* are specified by providing the constraints that needs to be applied on the static analysis prediction output. The constraints specifies the output files that needs to be selected for analysis and the filter criteria that needs to be applied for each of the output files.
2. For each run, the *Prediction filter constraints* file is generated, which is the primary input for the *Static Analysis Prediction Scanner*.

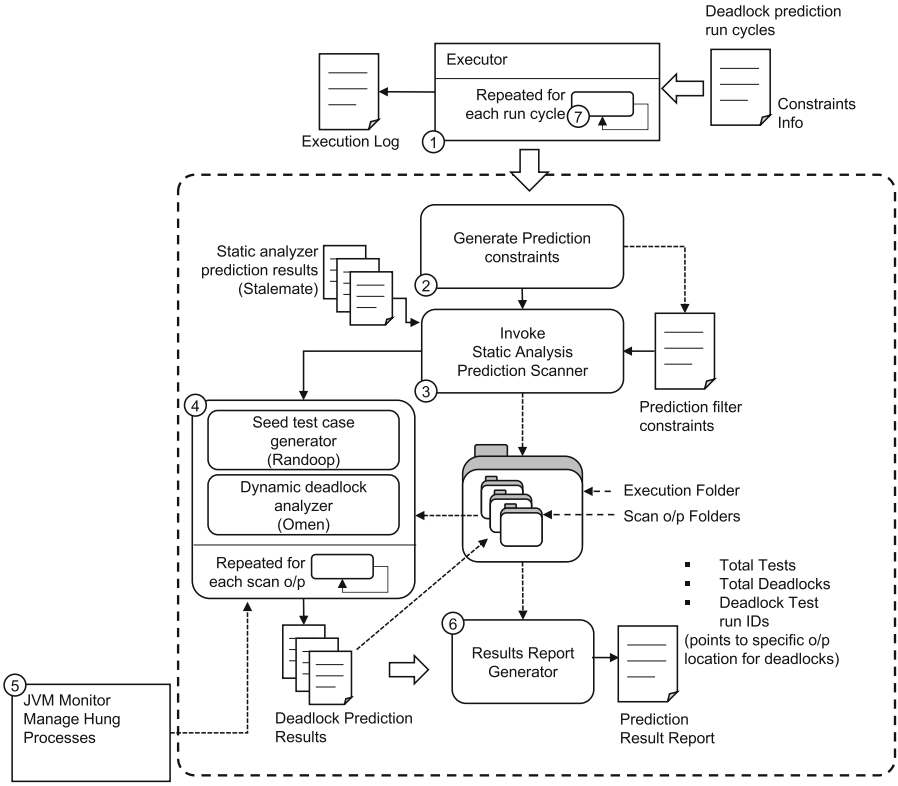

Fig. 6. Automated deadlock analyzer implementation design

3. The *Static Analysis Prediction Scanner* takes the generated Prediction filter constraints and scans each of the static analysis output files and selects the specific prediction nodes (the xml node that contain the lock order violations) along with the relevant metadata.
4. *Dynamic Deadlock Analyzer* is then invoked for each of the shortlisted node after generating the seed test cases for the classes involved in the potential deadlocks identified by the static analysis. The detected deadlocks are recorded in such a way that the corresponding static analysis prediction and the interleaving call details can be traced back.
5. During the execution of the dynamic analysis the *JVM Monitor* is invoked to monitor the execution for any hung scenarios. When such a scenario is detected, the monitor will extract the details for analysis and terminate the hung program so that the execution can continue. All the logs related to the deadlocking scenario are organized so that they can be seen in the context of the specific deadlock prediction.
6. After the completion of the analysis, the *Results Report Generator* is run to consolidate the complete execution results. The summary of each execution

is created in a master report file with the details of the constraints specified along with the results. With this information any detected deadlock can be consistently recreated.
7. The steps 2 through 6 are repeated for each of the run cycles specified in Step 1, thereby enabling the execution of a complete batch of automated analysis

The details of some of the key components of the implementation are described below:

Static Analysis Prediction Scanner. The static analysis prediction scanner is designed to shortlist the specific nodes from the output of the static analysis based on the constraints such as *Cycles*: Specifies the number of locks involved in a lock order violation, *Call Depth*: The number of function calls involved in a deadlock cycle that was predicted, *Call Density*: The number of classes involved in the prediction node resulting in a deadlocking cycle, and *Package Exclusions*: The list of packages that are to be excluded in the scan. All the necessary metadata required to target only the classes that could lead to a potential deadlock are collected at this stage. The Dynamic Deadlock Analysis is triggered after the completion of this scan.

Dynamic Deadlock Analysis. The first step of the process is to generate the sequential test cases that will act as the seed test cases for the dynamic analysis. Our aim is to target only the classes that are involved in a lock order violation, as predicted by the static analysis. The test cases corresponding to each of the selected prediction node from the static analysis are generated by the *Randoop* tool. *Omen*, is then invoked with the seed test case as input to initiate the dynamic analysis. The test cases are executed by Omen and the execution traces are scanned for cycles to detect the deadlocking scenarios. The detected deadlocks are then consolidated into a comprehensive report.

JVM Monitoring Routine. JVM Monitoring Routine was developed to handle the deadlocked analysis that happens either during *Randoop* execution or during *Omen* execution. JVM Monitor, once initiated is designed to run as long as there is an active JVM. As a first step, it fetches all the process identifiers (PIDs) of the java programs that are being executed in the JVM with a time delay. Then it checks if there are any identical processes between those time delays. If there are any identical processes, their respective stack traces are fetched. The stack traces are then compared to check the JNI global references held by the JVM. Analysis of the JNI global references is a predictable indicator to assess if the JVM is active or hung. If the JNI global references are identical across multiple snapshots with significant time delays, it predictably indicated a hung JVM. Once the hung state of the JVM is detected then logs are generated, and the details are collected in the reports. The processes that are hung are then terminated for the flow of the Dynamic Analyzer to progress ahead.

Report Generation. Following are the key reports generated during the execution:

1. **Test Runs Report:** Contains the execution status for each of the batches, along with the filter criteria applied for the analysis. This report provides an overview of the batch executions and enables to plan further batches for analysis.
2. **Execution Summary Report:** This report provides the summary of the execution results and helps to navigate to the specific deadlocking scenario. The metadata captured by this report enables to locate the specific Deadlock Report file along with the other details that indicates the number of tests executed and number of dead locking scenarios detected.
3. **Deadlock Report:** The Deadlock Report contains the log related to each of the nodes that were processed and which of the processed node resulted in a deadlock. If the dead locks are detected as a result of the hung JVM detected during test case generation or dynamic analysis, they are marked.

4 Results

The entire Java Runtime Libraries JRE v1.7.0_79 libraries (rt.jar) were subjected to Stalemate [1], the static analysis tool. The output files from the static analysis method were used as a starting point for the analysis. Table 1 lists the key details of the execution. We have been able to uncover deadlock in Java libraries that have not been demonstrated by other methods. We identified such cases and reported some of them to Oracle as bugs. Table 2 lists the bugs reported to Oracle.

Table 1. Summary of execution

Platform	MacOS High Sierra 17.3.0 Darwin Kernel Version 17.3.0, xnu-4570.31.3 1/RELEASE_X86_64 x86_6	
Java environment	java version "1.7.0_79", Java(TM) SE Runtime Environment (build 1.7.0_79-b15), Java HotSpot(TM) 64-bit server VM (build 24.79-b02, mixed mode)	
Execution summary	Total tests executed	1,563
	Total prediction nodes analyzed	26,728
	Duplicates eliminated	3,001
	Nodes filtered from static analysis	1,563
	Total deadlocking scenarios detected	**113**

Table 2. Bugs reported to Oracle

Deadlocking Java Library classes	Oracle JDK Bug ID
java.util.logging.Logger java.util.logging.LogManager	8194918
ava.awt.EventQueue sun.awt.AppContext javax.swing.plaf.basic.BasicDirectoryModel sun.awt.X11.XToolkit java.util.Vector	8194407
java.awt.EventQueue sun.awt.PostEventQueue java.awt.SentEvent sun.awt.SunToolkit	8194635
ava.awt.Component javax.swing.JFileChooser javax.swing.SwingUtilities java.awt.dnd.DropTarget javax.swing.JComponent	8194862
java.io.ObjectInputStream java.awt.KeyboardFocusManager java.awt.Component java.awt.Window java.awt.Frame	8194920
java.util.TimeZone java.util.Properties javax.naming.spi.DirectoryManager java.lang.SecurityManager java.util.Hashtable	8194919
java.awt.EventQueue java.awt.EventDispatchThread sun.awt.PostEventQueue	8194962
ava.awt.EventQueue sun.awt.AppContext javax.swing.plaf.basic.BasicDirectoryModel sun.awt.X11.XToolkit	8194983

Table 3. Examples: deadlocking call cycles

Calls between:	
`java.util.logging.Logger`	
`java.util.logging.LogManager`	
Forward calls	
`Logger.getAnonymousLogger()`	calls
`LogManager.getLogger(String)`	
`Logger.getLogger(String,String)`	calls
`LogManager.getLogger(String)`	
`Logger.getLogger(String)`	calls
`LogManager.addLogger(Logger)`	
Reverse calls	
`LogManager.addLogger(Logger)`	calls
`Logger.getLogger(String)`	
Calls between:	
`sun.awt.PostEventQueue`	
`java.awt.EventQueue`	
Forward calls	
`PostEventQueue.flush()`	calls
`EventQueue.postEventPrivate(AWTEvent)`	
Reverse calls	
`EventQueue.removeSourceEvents(Component)`	calls
`java.awt.SentEvent.dispose(AppContext,SentEvent)`	calls
`PostEventQueue.postEvent(SentEvent)`	
`EventQueue.push(EventQueue)`	calls
`EventQueue.getNextEvent()`	calls
`SunToolkit.flushPendingEvents()`	calls
`PostEventQueue.flush()`	
Calls between:	
`sun.rmi.server.Activation`	
`sun.rmi.server.Activation$GroupEntry`	
Forward calls	
`Activation.addLogRecord(Activation$LogRecord)`	calls
`Activation$ActivationSystemImpl.shutdown()`	calls
`Activation.checkShutdown()`	calls
`Activation$GroupEntry.restartServices()`	calls
`Activation$GroupEntry.getInstantiator(ActivationGroupID)`	
`Activation.addLogRecord(Activation$LogRecord)`	calls
`SystemImpl.shutdown()`	calls
`Activation$GroupEntry.unregisterGroup()`	
Reverse calls	
`Activation$GroupEntry.setActivationGroupDesc(ActivationGroupID)`	calls
`Activation.addLogRecord(Activation$LogRecord)`	

The results identified by our approach are reproducible. The static analysis [1] alone detected many thousands of potential deadlocks where one has to analyze the predictions manually to construct the deadlocking test cases, whereas we produce the deadlocking scenarios as output. The dynamic analysis [4] results are limited by the test cases that are subjected to it, hence it is not a viable tool in itself to analyze large libraries. Combining both together we have demonstrated an automated solution that is scalable for large libraries.

The Table 3 shows few examples of interleaving call details of the deadlocks detected by our method. For brevity the package names and return values of the methods are omitted while representing the call flows.

5 Conclusions

From the above results we assess that the method described was able to deal with the scale what it was intended for. It was successfully able to scan through tens of thousands of potential deadlocking scenarios and created over hundred reproducible deadlocking test cases without any false positives.

The approach was able to overcome the inherent limitation of the static approach of producing numerous 'potential' dead locking cases containing lot of false positives. Dynamic analysis on the other hand is effective in deadlock detection for programs for applications but not for libraries. It is limited by the coverage provided by the seed test case that is subjected to the analysis.

The presented approach provides an effective way to leverage both static and dynamic analysis methods to produce a viable automated way to detect deadlock in large java libraries.

Acknowledgements. We sincerely thank Dr. Murali Krishna Ramanathan for the discussion in formulating the problem. We also thank Malavika Samak and Dr. Murali Krishna Ramanathan for permitting us to use the program developed by them for dynamic deadlock detection.

References

1. Shanbhag, V.K.: Locating lock order violations in Java libraries - a scalable static analysis. Ph.D. dissertation. IIIT - Bangalore, Bangalore, Karnataka, India (2015). Reference [2] is the preliminary work of this thesis. Contact: IIIT-B Library (iiit-blibrary@iiitb.org) or Author (vivek.shanbag@gmail.com)
2. Shanbhag, V.K.: Deadlock-detection in Java-library using static-analysis. In: 2008 15th Asia-Pacific Software Engineering Conference, Beijing, 2008, pp. 361–368 (2008). https://doi.org/10.1109/APSEC.2008.68
3. Pandey, S., Bhat, S., Shanbhag, V.: Avoiding deadlocks using stalemate and Dimmunix. In: Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014), pp. 602–603. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2591062.2591136

4. Samak, M., Ramanathan, M.K.: Multithreaded test synthesis for deadlock detection. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2014), pp. 473–489. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2660193.2660238
5. Nagarakatte, S., Burckhardt, S., Martin, M.M.K., Musuvathi, M.: Multicore acceleration of priority-based schedulers for concurrency bug detection. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012), pp. 543–554. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2254064.2254128
6. Eslamimehr, M., Palsberg, J.: Sherlock: scalable deadlock detection for concurrent programs. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), pp. 353–365. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2635868.2635918
7. Jula, H., Tralamazza, D., Zamfir, C., Candea, G.: Deadlock immunity: enabling systems to defend against deadlocks. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI 2008), pp. 295–308. USENIX Association, USA (2008)
8. Jula, H., Tözün, P., Candea, G.: Communix: a framework for collaborative deadlock immunity. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN), Hong Kong, pp. 181–188 (2011). https://doi.org/10.1109/DSN.2011.5958217
9. Pradel, M., Gross, T.R.: Fully automatic and precise detection of thread safety violations. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012), pp. 521–530. . Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2254064.2254126
10. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA 2007), pp. 815–816. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1297846.1297902
11. Choudhary, A., Lu, S., Pradel, M.: Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, pp. 266–277 (2017). https://doi.org/10.1109/ICSE.2017.32
12. Naik, M., Park, C.-S., Sen, K., Gay, D.: Effective static deadlock detection. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), pp. 386–396. IEEE Computer Society, USA (2009). https://doi.org/10.1109/ICSE.2009.5070538
13. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006), pp. 308–319. Association for Computing Machinery, New York (2006). https://doi.org/10.1145/1133981.1134018
14. Joshi, P., Park, C.-S., Sen, K., Naik, M.: A randomized dynamic program analysis technique for detecting real deadlocks. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), pp. 110–120. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1542476.1542489

15. Williams, A., Thies, W., Ernst, M.D.: Static deadlock detection for Java libraries. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 602–629. Springer, Heidelberg (2005). https://doi.org/10.1007/11531142_26
16. Cai, Y.: A dynamic deadlock prediction, confirmation and fixing frame- work for multithreaded programs. In: Doctoral Symposium of the 26th European Conference on Object-Oriented Programming (ECOOP 2012, DS) (2012)