LOCATING LOCK ORDER VIOLATIONS IN JAVA LIBRARIES – A SCALABLE STATIC ANALYSIS

Vivek K. Shanbhag

Doctor of Philosophy Thesis

June 2015

International Institute of Information Technology, Bangalore

LOCATING LOCK ORDER VIOLATIONS IN JAVA LIBRARIES – A SCALABLE STATIC ANALYSIS

Submitted to International Institute of Information Technology,
Bangalore
in Partial Fulfillment of
the Requirements for the Award of
Doctor of Philosophy

by

Vivek K. Shanbhag
PH2006905

International Institute of Information Technology, Bangalore
June 2015

Dedicated

In the memory of my father

who told me, years ago, that I needed to pursue a doctorate;

And to my mother

for believing, that I would nevertheless get around to it, someday!

Thesis Certificate

This is to certify that the thesis titled **Locating Lock Order Violations in Java Libraries – A Scalable Static Analysis** submitted to the International Institute of Information Technology, Bangalore, for the award of the degree of **Doctor of Philosophy** is a bona fide record of the research work done by **Vivek K. Shanbhag**, **PH2006905**, under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

_____

Dinesha K. V.

Bangalore,
The 29th of June, 2015.

LOCATING LOCK ORDER VIOLATIONS IN JAVA LIBRARIES – A SCALABLE STATIC ANALYSIS

Abstract

Executions of some well-formed concurrent Java programs can sometimes surprisingly deadlock. Violation in the order of lock-acquisition by concurrent Java threads on shared objects can cause such deadlocks. Java designers, being well aware of this possibility, advised the use of *Resource Ordering* as a design technique to avoid such deadlocks. Unfortunately, the Java programming tool-chain failed to include any tool to help programmers ensure resource ordering. Consequently, even the extensively used Standard library exhibits large incidence of resource-order violations. In the present work we propose an analysis to detect lock order violations in large libraries. We have chosen the JRE as our case-study, and based upon our proposal, we have developed a prototype implementation of the analysis called 'Stalemate'. It conducts a global OO type based static analysis of the JRE, collecting and propagating runtime types to represent behaviours that would result from dynamic dispatch, to identify the set of reachable stack configurations corresponding to every publicly visible entry into the JRE. Stack configurations along which multiple locks are acquired can imply an ordering among their corresponding types. Given a set of stacks, if the order they imply on the types results in a cycle, then they violate type ordering. Stalemate reports all *sets of stacks* that each amounts to violation of type ordering. For the JRE, version 1.6, Stalemate reports 185,000 such sets involving less than 10% (1610 out of 16,500) classes in the JRE.

To demonstrate the usefulness of the analysis for library programmers, we examined a subset of the lock order violations and in 15 cases we could develop programs whose

executions could realise the violation resulting in deadlock. Two of these programs involve 3 threads deadlocked on 3-locks. One of the stacks includes a dynamic method dispatch that conventional static analysis, based upon compile-time type information, would not predict. Surprisingly many violations involve classes from earlier versions calling into later versions of the JRE. To the best of our knowledge, investigation on inputs as large as the JRE leading to discovery of such deadlocks has not been reported before.

To demonstrate the applicability of the analysis for users of Java programs, we successfully explored its use to enhance the usability and effectiveness of *Dimmunix*: a tool that can help avoid programs from repeatedly getting into known deadlocks.

Acknowledgements

I am most grateful to Professor K. V. Dinesha, for his guidance, support and encouragement. It was his strong and continuous support over the past many years that helped the completion of my thesis. Many parts of my work are influenced by his wise direction and his patience in reviewing different versions of my writing. His thoughtful guidance has always helped to direct the investigation on the right track. I am also grateful to Prof. PCP Bhat for playing the key role in helping launch my Doctoral pursuit. Without his timely and un-hesitant encouragement, I possibly would not even have started.

Special thanks to Dr. Srinivas Gutta for his magnanimous support for my Doctoral pursuit. I am also grateful to him for assigning me onto the 'insect repellent lighting' project that re-introduced me to the joys and pleasures of 'tinkering'[1].

I am forever indebted to Dr. K Gopinath, my Masters' thesis guide from whom we learnt a lot about computers and systems programming. His enthusiasm and passion for the subject, and his transparent style of discussing/communicating it during classroom sessions and in the CASL presentations has been instrumental in helping us (fortunate ones) to become ourselves.

I am thankful to my colleagues at my various places of work: My colleagues at the erstwhile Sun Microsystems: Thilak, Poonam, ShreeKumar and others; Balki, Purnendu, Srihari, and others at Philips Research have been very helpful. I am thankful to Divya, Guy Level, and others, during my association with Praxis High Integrity Systems.

I would like to acknowledge interesting and useful technical exchanges with William

[1]The website: www.exploratorium.edu defined 'tinkering' as "what happens when you try something you don't quite know how to do, guided by whim, imagination, and curiosity. When you tinker, there are no instructions – but there are also no failures, no right or wrong ways of doing things. It is about figuring out how things work and reworking them. Tinkering is, at its most basic, a process that marries play and enquiry."

Thies and Prof. Muralikrishna and Prof. Gopinath on the problem. I take this opportunity to thank the many reviewers who read the papers we submitted and provided very useful input. Many thanks to Srihari, Rajesh, and Manoj for reviewing different versions of our writing.

I am thankful to all members of team – 'Aloo-Gobhi.Com', for the wonderful understanding and co-operation within ourselves that has afforded me the bandwidth necessary for this pursuit. Thanks, Golu, for all the wonderful beer-sessions we have shared: May the beer keep flowing. Thanks to all my PhD colleagues at our research scholars room at IIIT-B, for keeping company: To Sita and Gopal for all those shared lunches and Tea-sessions.

Although, I may not have named everyone who contributed during my stay at IIIT-B, they will always be on my mind and my deepest appreciation goes to those who are not named here but played a positive role over the past many years.

I am grateful to the family support from both sides: parents, Smita, Sameer, Dinesh, Sushmita, Vayur & Meha and also my in-laws, Reshma, Shivu, Atharv, Athreyee, Raksha, Sachin, Viha, Guru, Vaishali, Krish-Maam, and the larger gang. Thanks, one and all.

Last, but not the least, I would like to thank, for everything that really matters, my wife Roopa, and our son Vidit, without whose unlimited love and endless support this work would not have been possible.

List of Publications (July 2006 -Aug 2015)

Papers related to the Thesis

1. Surabhi Pandey, Sushanth Bhat and Vivek K. Shanbhag, *Avoiding deadlocks using stalemate and dimmunix*, 36th International Conference on Software Engineering, ICSE'14, Companion Proceedings, Hyderabad, May June'14.
2. Vivek K. Shanbhag, *Deadlock-Detection in Java-Library Using Static-Analysis*, 15th Asia-Pacific Software Engineering Conference (APSEC 2008), December 2008, Beijing, China.
3. To be communicated: Vivek K. Shanbhag, Dinesha K. V, *Stalemate: A Global OO Type Based Static Analysis for Java*. Copies will be made available upon request.

Papers not related to the Thesis

1. Vijaykumar Channakeshava, Vivek K. Shanbhag, Avinash Panigrahi, Rajendra Sisodia and Sala Lakshmanan, *Safe subset-regression test selection for managed code*, Proceeding of the 1st Annual India Software Engineering Conference, ISEC 2008, Hyderabad, India, February 19-22, 2008
2. Vijaykumar Channakeshava, Sala Lakshmanan, Avinash Panigrahi and Vivek Shanbhag, *ChiARTS - Safe Subset-regression Test Selection for C#* Proceedings of the Fourth IASTED International Conference on Advances in Computer Science and Technology, ACST '08.
3. Vivek K Shanbhag and Harsh Dhand, *Insect Repellent Lighting: Design and Function*, Company Confidential Tech. Note # PR-TN2007/00339, June 2007.
4. Vivek K. Shanbhag, *Working IEEE/IFIP Conf. on S/w Arch. (WICSA) - A Visit Report*, Tech. Note # PR-TN2007/00524, Dec 2007.

5. Srinivasa Rao K, Anjaneyulu Pasala, Srinivas Gunturu, Arnab Datta Gupta and Vivek K Shanbhag, *Minimisation of Reg. Testing upon deployment of COTS-Updates*, Tech. Note # PR-TN2006/00637, Dec 2006.

6. Vivek K. Shanbhag and Udaysingh S. Patil, *Free-Source IT-Solutions for the SME-Segment – A case-study report*, Presented at NICOM 2007, Ahmedabad, Jan 2007, Published in Enhancing Enterprise Competitiveness (Marketing, People, IT, and Entrepreneurship)

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

The first version of Java was made available to programmers in the 1996-97 time-frame. Java was among the first[1] commercial languages to have Language level *support* for multi-threading. Traditionally[2], the three essential aspects of the Java programming framework that contributed towards this support for multi-threading were the following:

- The `java.lang.Thread` class from the Java Standard API: Using this API, the application could create multiple threads of control, assign them the program logic for execution, and control their execution in various ways.
- The Language keyword, `synchronized`: The primary intent for providing this facility was that the programmers can protect objects from non-constructive concurrent access from all other threads when one of them was in the midst of executing a piece of code that would update its state. The Compiler (`javac`) would translate such information provided by the programmer, appropriately, into advice for the Java Virtual Machine (JVM) that had the responsibility of enforcing its semantics during execution.
- The JVM: It ensures the monitor enter/exit semantics of `synchronized` pieces of code, the re-entrance for held locks, and the mutual exclusion for contested

[1] Earlier Languages including C [1] and C++ [2] hadn't supported multi-threading from their beginning. Support for the same was later retro-fitted into their scheme of things in the form of the Posix pthreads [3] library, for instance.

[2] Notably, over time, the semantics associated with all the three of them have evolved or changed.

locks. It accomplishes all these while employing an eager policy: On faced with a request for some lock by the code executing on some thread stack, the JVM immediately grants it if it is available or queues the request blocking thread execution, otherwise. Similarly, when faced with the last request to relinquish some lock from a thread stack, it does so, while additionally enabling one of the threads queued upon it to proceed.

The above organisation of the essential requirements that provision for multi-threading have worked quite well in the vast majority of use cases. In particular, the declarative fashion in which `synchronized` is used to express access control is particularly convenient and appealing. However, within such a block of code, when another method is invoked that itself also is `synchronized`, then the nested acquisition of the second lock while holding on to the first lock, imposes a 'lock ordering' on the two objects, or their types (respectively) as the case may be. Elsewhere if another functionality is programmed to acquire the same two locks (or their types) in the inverted order, then we have a lock order violation with potential for deadlock. Such usage of `synchronized` can lead to deadlocks, during execution. The purpose of this thesis is to investigate Library implementations with a view to flag such program fragments, and bring them to the attention of the concerned programmers. The next section uses a real example program to illustrate such problematic usage.

1.1 Motivation: A deadlocking Java Program

Consider the Java program: `DeadlockTest.java`[3], below. It is a good example of a (otherwise) well-formed multi-threaded Java application, whose execution under the JVM (version 1.4.2_04) leads to deadlock. Its execution starts two threads: each thread's `run()` method only enquires into the API. They do not (intend to) modify the

[3]Compiled & executed by: `javac DeadlockTest.java; java DeadlockTest`

state of any shared object. One thread enquires on the `TimeZone` API but discards the return value, and the other thread prints the list of all system properties and their current values onto standard output.

”DeadlockTest.java” ≡

```
public class DeadlockTest  {
  public static void main(String[] args)  {
    new Thread() {
      public void run() {
        java.util.TimeZone.getTimeZone("PST"); };
    }.start();
    new Thread() {
      public void run() { try {
        System.getProperties().store(System.out, null);
      } catch (java.io.IOException e) {
        System.out.println("IOException : " + e); }
      };
    }.start();
  }
}
```

Upon execution, the above program creates an object of type `DeadlockTest`, and causes the invocation of its `main()` method. Subsequently, the two objects of types (appropriately subclassed from) `java.lang.Thread` are instantiated, and respectively, their `start()` methods are invoked; Therefrom, their respective `run()` methods, defined above, are concurrently invoked in the two concurrent thread stacks. Such execution can lead to a deadlock. Once deadlocked, the JVM reports its thread state when prompted with the “ctrl+break” key sequence, as (partially) reproduced below[4].

```
Full thread dump Java HotSpot(TM) Client VM (1.4.2_04-b05 mixed mode):
[...]
Found one Java-level deadlock:
============================
"Thread-1":
  waiting to lock monitor 0x09b8c47c (object 0xaf0f4f10, a java.lang.Class),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x09b8c444 (object 0xab000560, a java.util.Properties),
  which is held by "Thread-1"

Java stack information for the threads listed above:
```

[4]Complete details of this bug used to be available at: http://bugs.sun.com/view_bug.do?6232022 or 6199320

```
=================================================
"Thread-1":
    at java.util.TimeZone.getDefault(TimeZone.java:498)
    - waiting to lock <0xaf0f4f10> (a java.lang.Class)
    at java.text.SimpleDateFormat.initialize(SimpleDateFormat.java:500)
    at java.text.SimpleDateFormat.<init>(SimpleDateFormat.java:443)
    at java.util.Date.toString(Date.java:981)
    at java.util.Properties.store(Properties.java:531)
    - locked <0xab000560> (a java.util.Properties)
    at DeadlockTest$2.run(DeadlockTest.java:9)
"Thread-0":
    at java.util.Hashtable.get(Hashtable.java:332)
    - waiting to lock <0xab000560> (a java.util.Properties)
    at java.util.Properties.getProperty(Properties.java:563)
    at java.lang.System.getProperty(System.java:575)
    at sun.security.action.GetPropertyAction.run(GetPropertyAction.java:66)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.util.calendar.ZoneInfoFile.readZoneInfoFile(ZoneInfoFile.java:900)
    at sun.util.calendar.ZoneInfoFile.createZoneInfo(ZoneInfoFile.java:520)
    at sun.util.calendar.ZoneInfoFile.getZoneInfo(ZoneInfoFile.java:499)
    - locked <0xaf0f9380> (a java.lang.Class)
    at sun.util.calendar.ZoneInfo.getTimeZone(ZoneInfo.java:531)
    at java.util.TimeZone.getTimeZone(TimeZone.java:448)
    at java.util.TimeZone.getTimeZone(TimeZone.java:444)
    - locked <0xaf0f4f10> (a java.lang.Class)
    at DeadlockTest$1.run(DeadlockTest.java:5)

Found 1 deadlock.
```

The above deadlocked execution of `DeadlockTest` contains application threads `Thread-0` and `Thread-1` that are both blocked, awaiting the acquisition of a lock held by the other. The `run()` on `Thread-0` starts by invoking the `getTimeZone()` method, which, being `static synchronized`, locks the `java.lang.Class` object corresponding to its Class: `java.util.TimeZone`. The `run()` method on `Thread-1` invokes the `store()` method on a globally assessable `java.util.Properties` object. `java.util.Properties.store()` being a `synchronized` method, locks the `Properties` instance that is target of the method invocation. In this way, both the threads acquire one lock each, on objects that are accessible to computations on both the thread stacks.

The computation on `Thread-0` proceeds, through nested method invocations, to eventually invoke the `getProperty()` method on the same `Properties` instance that

is already locked by `Thread-1`. This method is not synchronized. However, the `java.util.Properties` class is a subclass of `java.util.Hashtable` class which defines a `synchronized` instance method called `get()`. The `getProperty()` method invokes the so inherited `java.util.Hashtable.get()` method, whose execution attempts to, firstly, lock the target object of type `java.util.Properties`. This attempt blocks awaiting its release by `Thread-1`.

The computation on `Thread-1` proceeds, through nested method invocations to eventually invoke the `static synchronized` method `java.util.TimeZone.getDefault()`. Which execution, similarly, blocks awaiting the release of the lock acquired by `thread-0` on the `java.lang.Class` object corresponding to `java.util.TimeZone` class.

The three objects (attempted to be) locked by the 5 occurrences of the `synchronized` usage in the above two thread stacks are all objects that are created by or on behalf of the library classes: the two Class objects corresponding to the `TimeZone` and the `ZoneInfoFile` classes are created during class loading. The object of type `java.util.Properties` is actually an class attribute of the `java.lang.System` class that it exports/shares via the `getProperties()` method. None of the locks involved in the above deadlock are created by the application code.

It is useful to note, also the reasons, why, the developers chose to tag certain methods or statement blocks as `synchronized`. In the two thread stacks above, the five locks acquired are discussed below:

- The `static` methods `getTimeZone()` in file: `TimeZone.java` may call into code that initialise the value to be returned from accessing into the configuration files. The developers may want to serialise such access for correctness or performance reasons.

- The `static` method `getZoneInfo()` in file: `ZoneInfoFile.java` may call into

code that initialise the value to be returned from accessing into the configuration files. The developers may want to serialise such access for correctness or performance reasons.

- The `getProperty` instance method of the `Properties` class invokes its `get` method that it actually inherits from the parent class `HashTable`. The `get` method is `synchronized` to ensure that the key value association cannot be modified while it is being accessed.

- The `store` instance method of the `Properties` class merely writes the text serialised rendering of the list of properties onto the `PrintStream` object handed in as argument. It locks the `Properties` object to ensure that its state is not updated while its contents are still being serialised out.

- Finally, the `static` methods `getDefault()` in file: `TimeZone.java` may call into code that initialise the value to be returned from accessing into the configuration files. The developers may want to serialise such access for correctness or performance reasons.

While all the justifications for the use of `synchronized` seem reasonable, the fact that they lead to a deadlocks is somehow not good. The above bug was fixed by making the following correction in the file: `Properties.java`.

```
/tomcat/webapps/ctetools/CodeStore/1298/webrev/src/share/classes/java/util/Properties.java-
Wed Dec 22 20:30:31 2004
--- Properties.java     Thu Dec  2 20:34:03 2004

*** 523,533 ****
       */
!     public synchronized void store(OutputStream out, String header) throws IOException
      {
          BufferedWriter awriter;
          awriter = new BufferedWriter(new OutputStreamWriter(out, "8859_1"));
          if (header != null)
              writeln(awriter, "#" + header);
          writeln(awriter, "#" + new Date().toString());
          for (Enumeration e = keys(); e.hasMoreElements();) {
              String key = (String)e.nextElement();
```

```
--- 523,534 ----
       */
!     public void store(OutputStream out, String header) throws IOException
      {
          BufferedWriter awriter;
          awriter = new BufferedWriter(new OutputStreamWriter(out, "8859_1"));
          if (header != null)
              writeln(awriter, "#" + header);
          writeln(awriter, "#" + new Date().toString());
+         synchronized (this)  {
              for (Enumeration e = keys(); e.hasMoreElements();) {
                  String key = (String)e.nextElement();

*** 540,544 ****
--- 541,546 ----
                  val = saveConvert(val, false);
                  writeln(awriter, key + "=" + val);
              }
+         }
          awriter.flush();
      }
```

The execution of `DeadlockTest` using JVM Version 1.4.2_05, for instance, does not deadlock as above. Notice that the correction reduces the scope of the `synchronized` portion of the processing in the `Properties.store` method. More specifically, the correction moves the call to `Date.toString()` outside the scope of the `synchronized` statement thus rendering the thread stack for "Thread-1", above, unreachable; thereby averting the deadlock.

In this thesis we develop a scalable static analysis for the Java language that can predict the above class of problems in libraries or applications. Also a prototype implementation of the same is developed for libraries that is used to analyse the version 1.6 of the entire Java standard library. Based upon and starting from the output of the analysis we have also developed 15 small programs that can indeed deadlock the JVM as predicted by the analysis. 10 of these bugs are also present in the version 1.7 of the Library.

1.2 A bug classification perspective

In general, the ability to write Java programs whose executions may deadlock due to shared locks being taken in cyclic order is not surprising: Section 10.7 of [4] discusses such 'Deadlocks' with an example, and observes[5]: "*You* are responsible for avoiding deadlock. The runtime system neither detects nor prevents deadlocks. It can be frustrating to debug deadlock problems, so *you* should solve them by avoiding the possibility in *your design*. One common technique is to use *resource ordering*. ...". However, when trying to understand this JVM design standpoint in the context of current Java programming practice, there are some interesting observations:

- *Which* programmer should attempt to avert an application program from reaching its deadlock. It could be the application programmer; it cannot be only the Library API programmer(s). But, ideally, it should include both. Unfortunately, there is no tool for programmers to use, yet, that would take a global perspective of all the relevant classes to enable the use of global *resource-ordering*. For a library as large as the JRE, whether/how such a tool can be built is also not clear. At what abstraction level can the *resource ordering* technique be applied to yield useful insights, given the industrial strength size of the JRE. This thesis describes *Stalemate*, a tool that addresses this need. The design choices leading to its design and implementations, and the results fetched therefrom are described in the rest of this thesis.

- Most Java applications resort to significant re-use of functionality offered by the library. When such applications do expose scenarios that involve deadlocks due to cyclical locks, it is important to use a reasonable basis to assign responsibility of fixing the code to avoid the possibility of deadlock. The rest of this section describes a bug classification system that may be used to evolve such a basis.

[5]Emphasis introduced by this author

Table TC1.1: Bug Classification type correspondence.

Objects		Types		*Responsible* Types	
t_1	Thread-0	Type of t_1	DeadlockTest$1	C_{t1}	DeadlockTest
t_2	Thread-1	Type of t_2	DeadlockTest$2	C_{t2}	DeadlockTest
l_1	Class object for java.util.TimeZone	Type of l_1	java.lang.Class	C_{l1}	ClassLoader
l_2	Properties attribute: java.lang.System.props	Type of l_2	java.util.Properties	C_{l2}	java.lang.System
m_{l1}	getTimeZone			C_{ml1l2}	java.util.TimeZone
m_{l2}	store			C_{ml2li}	java.util.Properties

Recollect from the discussion about `DeadlockTest` in section 1.1, that the three essential components of a deadlock are the threads, the objects being locked, and the code that is executing on the thread stacks. All of these are manifestations that may be created by classes (including their methods) developed either by the library programmer, or the application programmer. In Java, class is the smallest unit that can be completely provided by a programmer. We use this granularity to evolve a way to assign responsibility of fix.

Consider a deadlock involving threads t_1 & t_2, and locks l_1 & l_2. It can be argued that some methods of the classes C_{t1}, and C_{t2} which create threads t_1 & t_2 are the ones *responsible* for them. For the locks l_1, and l_2 it can be (similarly) argued that some methods of the classes C_{l1}, and C_{l2}, which create them are *responsible* for them. Consider, further, that method m_{l1} locks l_1 on thread t_1, and that method m_{l2} locks l_2 on thread t_2. (In order for this deadlock to be realised, t_1, while holding lock l_1 must block on lock l_2, and similarly, t_2, while holding lock l_2 must block on lock l_1). Let Classes C_{ml1l2} and C_{ml2l1} be the classes that define methods m_{l1} and m_{l2}, respectively. Table TC1.1, lists correspondingly the actual types involved in the `DeadlockTest` example program.

In general, it can be argued that the responsibility to *avoid by design* any reachable deadlock in an application, collectively rests with the providers of all the *responsible* types, as illustrated above. In case of `DeadlockTest`, observe from the entries in the last column of the Table TC1.1 that, 4 out of 5 *responsible* types are from the JRE.

It was therefore, only natural, that the Library programmers accepted responsibility for the bug, and fixed it. The perspective described above is merely indicative: in the presence of OO features like inheritance and polymorphism, trying to define a rigorous basis by arguing along the above lines requires more work.

For the purposes of this thesis we merely observe that if a significant subset of the responsible types are from the JRE, then such a deadlock may make a strong candidate requiring to be fixed by the Library programmers. Whereas, on the other hand, in particular, when the types responsible for the threads and the locks involved in the deadlock are from the application space while merely the `synchronized` code is from the Library, then, such deadlocks can be *worked around* using application level locking to serialise access to the shared locks.

1.3 Problem Statement

The motivating example, `DeadlockTest` is not a program that deliberately attempts to deadlock its own execution. It is a typical example of a deadlock wherein the objects locked, and the code that locks the objects are all provided by the library. The application classes merely create the multiple threads involved in the deadlock. In fact, the two threads do not even try to (obviously) modify the state of any objects shared by them. The program does the minimal work necessary to *uncover* a potential deadlock opportunity already present in the Library implementation. Therefore, to fix the problem, the Library needed correction.

It is our contention that reason why the Java Library contains many such deadlock opportunities is because the library designers and developers did not have any tool support to flag all the 'lock order violations' that crept into the code as it was developed and evolved over the years. The need for a static analysis of libraries with a view to identify sets of entry points, that, when executing concurrently on multiple threads of an ap-

plication may lead them to deadlock, remains unaddressed. The purpose of this thesis is to demonstrate that such an analysis can be conducted, even on industrial strength libraries like the JRE. The present thesis addresses the following aspects of the design of the tool:

Where: The intent of the tool is to identify lock order violations, which is a global property. We position the tool after the compilation of all the source and forming the `jar` file. By doing so, we ensure that the tool gets to inspect classes corresponding to only well-formed java.

What: are the specifications for the tool? We want the tool to conduct a static analysis similar to what the Compiler does, *i.e.* at the level of Types. The analysis must handle the entire Java language, and conduct an accurate analysis. In particular, it must precisely model both: dynamic dispatch and Exception handling. It may appear unusual how a static analysis can account for dynamic dispatch: However, collecting and propagating information regarding the runtime types of objects that are created and assigned, rather than the declared types of reference variables to which assignments are made, helps to achieve this.

How: Chapter 3 outlines our approach to meet the requirements.

1.4 Organisation of the Thesis

The rest of the thesis is organised as follows: Chapter 2 is literature survey. Chapter 3 gives an overview of the thesis work. Chapter 4 describes the structure and the construction of the Invocation Graph. Chapter 5 details the Abstract Interpretation of the method body. The TLOG and cycles therein that correspond to lock order violations and the corresponding output are discussed in Chapter 6. Chapter 7 discusses our results. Chapter 8 describes the use of the results of our analysis to enhance the useful-

ness and the effectiveness of a deadlock avoidance tool called Dimmunix. Finally our conclusions, contributions, limitations, and Future work are discussed in Chapter 9.

1.5 Appendix: Recap of Java related background

1.5.1 The SYNCHRONIZED keyword

`synchronized` is a keyword of the Java language. A complete description of, traditionally, the semantics associated with its usage is described by the Java Language Specification (JLS) [5]. Around 2004-05, further semantics was augmented to its usage along with the introduction of the Concurrency API in Java version 1.5 and the associated Java Memory Model. Below we briefly revisit the traditionally associated semantics with its usage.

In general, the `synchronized` keyword can be used to decorate an entire method body, or merely a smaller part of it, comprising of a statement block. It has declarative semantics, used to lock and unlock the object being operated upon.

Synchronized Methods: When used to decorate an entire method definition, depending upon whether (or not) the method is `static`, the lock is either an instance-lock or else it is a type-lock, (respectively). When used in this manner, `javac` merely decorates the header of the corresponding method body in the generated class file to indicate that it is `synchronized`. Subsequently when such a method is invoked, during program execution, the JVM notices the method decoration and acts upon it by ensuring that the target object of the method invocation (in the case of an instance method), or (otherwise) the `java.lang.Class` object corresponding to the containing class (in case of a `static` method) is locked before the method body is allowed to proceed with its computation. Further, upon completion of execution of the method body, the JVM arranges to relinquish the acquired lock. The method may terminate in the normal course of

action, for instance due to the execution of a `return` instruction, or it may be a due to the `throw` of an exception, for instance some runtime exceptional condition being triggered. In either case it is the responsibility of the JVM to ensure that the lock acquired prior to the computation of the method body is relinquished upon its exit.

Synchronized statements: `synchronized` can also be used inside the method body, as: `synchronized ( expr ) { statements }` to indicate that the enclosed *statements* are to be executed after acquiring a lock on the object fetched from evaluation of *expr*. When used in this manner, `javac` translates it into a sequence of bytecode instructions that contain a single occurrence of the `monitorenter` bytecode and (atleast one) some occurrences of the `monitorexit` bytecode. Upon encountering these `monitorenter/exit` set of instructions the JVM, correspondingly, acquires and releases the lock on the referenced object. The bytecode sequence corresponding to the evaluation of the *expr* precedes the `monitorenter` instruction. The bytecode sequence corresponding to the *statements* immediately follows it. Often, depending upon the structure of *statements*, there may be more than one `monitorexit` instructions that, all, match the same `monitorenter` instruction. Additionally, more (similarly matching) occurrences of `monitorexit` are inserted into the generated code to ensure that the said lock is relinquished also during exceptional control flow.

As described above, `javac` the compiler and the JVM, together co-operate to ensure that irrespective of how the control flows at runtime, the block structured semantics of the lock and unlock operations corresponding to the source code use of `synchronized` is honoured during execution of the corresponding bytecode.

Re-entrency: The locks so acquired by the JVM, as described above, are held on behalf of the threads. As implied by the block structured semantics of locking, when a `synchronized` instance method `Cr.sm1()` invokes another `synchronized` instance method (say) `Cd.sm2()`, then the second lock (on object of type `Cd`) is attempted to be

acquired while the first lock (on object of type `Cr`) is still held, by the corresponding thread. These locks are re-entrant: If a thread acquires a lock on object `ok`, and while holding it, invokes another of its `synchronized` methods that requires the same lock, it is automatically granted. The lock `ok` becomes available only after control exits from the outermost `synchronized` block that locked it.

The above description of the use of `synchronized` refers to the mutual exclusion aspect of its semantics: since only the thread that holds the lock must access its state, requiring all such access to be similarly `synchronized` causes the effect of serialising access from multiple threads.

Happens-before Relation: When an object is accessed from multiple threads, it is important to ensure that updates made from one thread are properly visible to the code from the other thread. This is also achieved through the use of the `synchronized` keyword. This additional semantics has become clearly defined as part of JSR-133 that specifies the Java Memory Model. The Java Memory Model FAQ [6] explains in simple words as: "Synchronisation ensures that memory writes by a thread before or during a synchronized block are made visible in a predictable manner to other threads which synchronise on the same monitor. After we exit a synchronized block, we release the monitor, which has the effect of flushing the cache to main memory, so that writes made by this thread can be visible to other threads. Before we can enter a synchronized block, we acquire the monitor, which has the effect of invalidating the local processor cache so that variables will be reloaded from main memory. We will then be able to see all of the writes made visible by the previous release." ... "it is important for both threads to synchronise on the same monitor in order to set up the happens-before relationship properly. It is not the case that everything visible to thread A when it synchronises on object X becomes visible to thread B after it synchronises on object Y. The release and acquire have to "match" (i.e., be performed on the same monitor) to have the right semantics. Otherwise, the code has a data race."

1.5.2 Some Background, tools, and Terminology

The Java Language is specified by the Java Language Specification (JLS) [5]. The Java compiler (`javac`) translates Java source into class files. The Class file format, the bytecode corresponding to executable statements in the java source, and its interpretation / execution is specified by the Java Virtual Machine Specification (JVMS) [7]. The Java standard library is specified (traditionally) through Java Documentation (Javadoc) pages, and (of-late) more formally through the JCP. It is offered in the form of the Java Runtime Environment (JRE), which includes the (multi-)threading API, `java.lang.Thread`.

The Java Standard Edition is a free offering from (erstwhile Sun Microsystems, later acquired, and currently owned, by) Oracle Systems that implements all of the above. It has since been made open and is referred to as the OpenJDK (Open Java Development (tool)Kit). More simply, the JDK comprises of the various Java tools and the JRE. The JRE contains the JVM and a bunch of *jar*-files that together implement the various APIs. The source for all of these is available as the OpenJDK.

Versions 1.3.1, 1.4.2, 1.5.0, 1.6.0 and 1.7.0 of the JDK are major releases of the technology, that have happened (approximately) 18 months apart; these are also referred to as the train releases, and they bundle along new functionality in the form of either larger APIs, or better engineered JVM, or (usually) both. Along each of the trains there are update releases (like 1.4.2_12 or 1.5.0_3, for instance). These update releases are typically once a quarter and they bundle along bug fixes, *along that train*. The API specification (to the extent possible) remains frozen along all update releases of a given train. Effectively the Java Standard API grows with every new major release. Sun Microsystems has encouraged users to remain current with the update release along whichever train they use, and also try hop onto a more recent train, when possible.

Table TC1.2: The History and growth of Java SDK.

API Train	FCS Ver. & date	Last Ver. & date	releases	APIs	Types	Now
1.1	1.1.6_009 dtd 10th June 1997	1.1.8_015 dtd 8th August 2002	10	22	1444	eol
1.2.[1,2]	1.2.1_004 dtd 11th Feb 1999	1.2.2_017 dtd 5th Sept. 2003	15	60	4929	eol
1.3.[0,1]	j2sdk1.3.0 dtd 31st Aug 2000	1.3.1_29 dtd 9th Sept. 2010	36	76	6204	eol
1.4.[0,1,2]	j2sdk1.4.0 dtd 30th Jan 2002	j2sdk1.4.2_30 dtd 3rd Feb. 2011	40	135	10839	eol
1.5	j2sdk1.5.0 dtd 15th Sept. 2004	j2sdk-5.0.u.22 dtd 9th Oct. 2009	23	166	15538	eol
1.6	javaSEDK6 dtd 29th Nov. 2006	javaSEDK6-u45 dtd 27th March 2013	39	203	20435	alive
1.7	javaSEDK7 dtd 27th June 2011	javaSEDK7-u40 dtd 27th August 2013	17	209	–?–	alive

Along every train of the JDK, (to the extent possible) the Class file format, of classes generated by the Java compiler, remains the same. The information in the Class file is binary encoded. The Java tool kit includes a tool called `javap`, that reads the class file, and renders its contents into ASCII-text. For instance, the bytecode generated by the compiler for some member method is stored as an sequence of binary encoded, ordered, set of instructions in the class file. `javap` reads such a class file, and prints out ASCII text rendering of the same. The snippet below is generated from the commandline: '`javap -c -private -verbose java.lang.Class`'. It contains the entire translation for its `getGenericSignature` method. Its return type is an array of `java.lang.reflect.Type`. Its invocation expects a single argument: the `this` pointer. The execution of its method body uses an operand stack of depth 1 word, and an local array with 1 entry. The contents of the bytecode array are elaborated in the form of an integer index into the array, and the bytecode stored at that location (onwards). Finally the correspondence between the line numbers in the source file, and the index into the bytecode array that marks the beginning of the corresponding portion of the bytecode is listed under `LineNumberTable`.

```
[...]
public java.lang.reflect.Type[] getGenericInterfaces();
  Code:
```

```
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #793; //Method getGenericSignature:()Ljava/lang/String;
   4:   ifnull  15
   7:   aload_0
   8:   invokespecial   #812; //Method getGenericInfo:()Lsun/reflect/generics/
                                    repository/ClassRepository;
   11:  invokevirtual   #925; //Method sun/reflect/generics/repository/ClassRepository
                                    .getSuperInterfaces:()[Ljava/lang/reflect/Type;
   14:  areturn
   15:  aload_0
   16:  invokevirtual   #781; //Method getInterfaces:()[Ljava/lang/Class;
   19:  areturn
  LineNumberTable:
   line 793: 0
   line 794: 7
   line 796: 15
[...]
```

1.5.3 Bytecodes monitorenter and monitorexit

The JVMS details the various bytecode instructions including the `monitorenter` and `monitorexit` that `javac` generates (appropriately) on encountering `synchronized` statements in method bodies in Java source. The JVMS also specifies the interpretation / execution of these bytecode instructions by the Java virtual machine. Below, in this section we merely illustrate the manner in which, for instance, the `monitor` instructions are generated by the Java compiler, in response to the programmers' use of `synchronized` statements in the Java Source. Consider the code pattern: "`synchronized (` *expression* `) {this.doThat();}`". Its translation into bytecode by `javac` whose text rendering by `javap` looks [6] like follows:

```
 ...:  ...          // code corresponding to the evaluation of 'expression'
 ...:  ...          //  its resultant reference is kept in locals array@index 3
  13:  aload_3      // load reference of object to be locked onto operand stack
  14:  monitorenter //    lock the target object
 ...:  ...;         // code corresponding to invocation of 'this.doThat()'
  23:  aload_3      // re-load reference of the locked object onto operand stack
  24:  monitorexit  // un-lock the target object
  25:  astore_2     // 'this.doThat()' threw some exception, store its reference
  26:  aload_3      // re-load reference of the locked object onto operand stack
  27:  monitorexit  // un-lock the target object
```

[6] Comments starting with '//' are inserted by this author to ease readability of the bytecode

```
  28:  aload_2      // re-load reference of the exception onto operand stack
  29:  athrow       // re-throw, causing exceptional termination of this invocation
 ...: ...
Exception table:
    From To Target Type
      15 25 25 any
      25 28 25 any
```

Notice in the above bytecode sequence, there is a single `monitorenter` but two `monitorexits`. The first one corresponds to the release of the monitor after normal execution of `this.doThat()`, and the second one corresponds to the release of the monitor in case any runtime exception occurs during the attempt to compute `this.doThat()`. So also, notice that the ”Exception table:” has two entries that have ’25’ as the target, and ’any’ as their ’Type’. The first of them encapsulates the entire body of the `synchronized` statement between its ’from’ and ’to’ bytecode offsets. More specifically: the ’from-value’ corresponding to this entry is always one greater than the bytecode offset of the `monitorenter` that marks the beginning of the encapsulated code, and the ’to-value’ is one greater than the bytecode offset corresponding to the encapsulated code end. Occasionally, when there is an ’if-then-else’ statement as the encapsulated code, wherein on one of the branches (either the then-branch or the else-branch) the processing terminates with a return of control to the calling method, then there can be more then two `monitorexit` instructions corresponding to the `monitorenter`.

It is also instructional to look at the two entries into the ”Exception table:”. The first one (in the above example, and in general, all but the last of them) corresponds to the ’normal-processing’ of the ’encapsulated-code’. Any exception occurring during normal-processing causes the control flow to get diverted into the ‘finally clause’ that ensures that the lock is released before the exception causes the call stack to unwind until there is a programmer provided catch clause for the specific type of exception thrown. The last entry, in general, provides for essentially the same need just in case some runtime exception gets thrown *during* the execution of the finally clause.

CHAPTER 2

LITERATURE SURVEY

Existance of deadlock opportunities in the Java Library is well known in the research community, and recently there is an interesting vareity of work reported about it. In general, this problem has been addressed using many different approaches: Amy, *et al* [8] and Jyotirmoy, *et al* [9] conduct static analysis of libraries, whereas Mayur Naik, *et al* [10] report static analysis of Java Programs. Bandera [11] extracts a model from the source for entire Java programs that is used to drive a model-checker to identify reachable deadlocks. Horatiu Jula, *et al* [12] and Fancong Zeng *et al* [13] report prototypes of approaches to avoid previously witnessed deadlocks during program execution. In our discussion, below, we look at static analysis methods in greater detail since ours is also such a method. In this chapter, our intent is to briefly describe the various other approaches used, in the literature, to address problems similar or related to ours. The discussion related to a comparison of our results with those of some of the others is in Section 9.2.

2.1 Static Analysis Approaches

In general, our observation is that most of the reported static analysis methods are based upon an underlying alias analysis approach. Their approach, used as the basis for their heap-model, by [8–10] is to start by modelling all objects allocated at a given

Table TC2.1: Empirically observed size of the input: JRE version 1.6

JRE Version: 1.6.0_35		Inspecting method bytecode	
Classes:	16,500	`new` bytecodes:	1,31,000
Interfaces:	2,300	`invoke` bytecodes:	6,06,000
Inheritance Relations:	35,000	Total `locals`:	4,29,000
Implements Relations:	22,000	SSA overhead:	2,10,000

allocation site in the code by a single abstract heap object. One of the early works along these lines by Amy, *et al*, [8], reports results from an analysis of about 1100 classes from the JRE 1.4.2: their library analysis based upon such modelling of the heap would become intractable. In order to better understand the reason why such large models are fetched from such modelling we conducted an empirical study of the size of the JRE. The results of this study are tabulated in Table TC2.1.

The left-hand column of Table TC2.1 reports the size of the JRE: the number of Classes, and interfaces it defines. The count (Class) 'Inheritance relations' is fetched from computing the transitive closure of the `extends` relations between Class definitions. Similarly, the transitive closure of the `extends` relations between Interfaces, and the `implements` relations between Classes and Interfaces is computed to fetch the count of 'Implements relations' of the type: Class `implements` Interface.

The right-hand column of Table TC2.1 reports accumulated counts of aspects that are discovered from inspecting the bytecode of all the methods defined by the JRE: the number of instances where `new` object(s) is(are) created, or where some method is `invoke`d. Similarly, the accumulated count of local variables over all method definitions. The SSA overhead is estimated by counting the number of instances of assignments into local variables. Finally, the number of method behaviours that may return objects of different types during their different invocations are also counted.

Grove *et al* [14]: This paper describes a lattice of models that may be computed from analyzing entire programs. All the analyses envisaged by their framework are

based upon alias analysis methods that represent all objects allocated at the same allocation site using at-least one symbolic heap object. In comparision, our analysis uses a single symbolic node to represent all objects of the same type. Our model is therefore more abstract than all the models described in this paper.

The analysis by Artho and Biere [15]: This paper describes, briefly the up-gradations to Jlint to fetch Jlint2, and the results of subjecting various large s/w packages to it. The paper uses ”Jlint1” to refer to the existing features of Jlint; we shall do the same. Jlint1 already included the ability to detect cycles in a lock graph: defined by the order in which threads access the locks. Based upon an empirical observation that about 55% of the use of ”synchronized” was at method-level (which Jlint1 already handled), and another 30% was at statement level that either aquired locks on ‘this’, or on some ‘constant’ field, the paper briefly hints at an approximate extension to the ”lock graph” that is able to model totally about 85% of the use of ”synchronized”. Their model is an approximate way to upgrade Jlint1 to Jlint2 with a view to maximize value while incurring minimal implementation cost.

Particularly interesting, in contrast to our assertion (in this thesis) that deadlock is a global property and its prediction requires that all library classes be included into the computation, is their observation: ”Method calls from synchronized blocks to other classes are not included in the call graph because the call graph would grow too big for the current implementation of Jlint. This restriction confines deadlock detection to deadlocks within and across methods of the same class. Jlint cannot detect deadlocks across different classes, except for deadlocks across synchronized methods. Also, inheritance is not fully covered, as the behavior of superclass methods is assumed to be consistent with inherited methods, with respect to synchronization.”

Our ”Invocation Graph” & ”Type Lock Order Graph” combine do not suffer any of the above limitations. Nevertheless, in their section 7.2, entitled ”Sun’s JDK packages”,

they state: Jlint2 as such is capable of analyzing large packages, such as Sun's JDK packages. However, the number of potential deadlock warnings was very high (several thousand). By filtering most warnings, the number could be reduced to a few hundred, which is manageable. Because the Java Foundation Classes are quite mature in version 1.3, ...": Way back in 2001 , they could detect, using approximate modelling, "several thousand" deadlock warnings. Furthermore, they were able to identify "filters" to help reduce the number of warnings from several thousand to a few hundred. It will be interesting to look at these filters and their applicability to the results of our analysis.

The analysis by Amy, *et al* [8]: This work is among the first attempts to analyse libraries with a view to predict reachable deadlocks. Their approach to avoid reporting cycles protected by gate locks (or fat locks) is interesting. It was useful in influencing our implementation to report fewer lock order violations as reachable concurrently.

Their analysis includes an augmentation to a compiler to generate method-level summaries in the form of a *symbolic state* for every method, that includes a lock order graph over nodes (of the form $< tName, allocationSite >$). Each such node represents a collection of heap objects, all of type *tName* that are allocated at the same site: *allocationSite*. Method calls are handled by integrating the graph for the called into that for the caller: since this operation can upgrade the summary for the caller method, a fixed point computation is necessitated to arrive at the final summary for each method of the library.

Using the numbers from Table TC2.1, for the JRE, alias analysis would fetch a total of over 560,000 (131,000 + 429,000) different nodes in the lock order graph for the entire library. Moreover, its way of handling "method calls" would require to make over 606,000 computations, that would evolve the summaries for 160,00 methods, followed by a fixed point computation until the summaries stabilise.

The analysis includes the use of a *join operator* "to combine states along confluent

paths of the program", that chooses "the strongest type constraint for the fresh object, ..., their lowest common super-class". In order to correct for this lossy "join operator", they include a post-processing step, since: "because the analysis for each method was based on the declared type of locks, extra edges must be added for *all possible concrete types* that a given heap object could assume".

JRE, version 1.6 defines over 16,700 classes, and over 2,200 interfaces, including close to 35,000 instances of the superclass-subclass relation, and over 22,000 instances of the class-implements-interface relation. After the fixed-point computation completes, their post-processing step has to be carried out for all of the 57,000 (35,000 + 22,000) relations. That is a lot of computation, on a very large model of the input.

The analysis by Lhotak and Hendren [16]: This paper reports a comparative study of various ways to represent the large number of calling contexts detected during a static alias analysis of a Java program. As indicated by its title, a points-to analysis essentially has to model pointer variables, and the target objects they may point to. Since the number of objects are typically too large, a standard abstraction is to represent all objects allocated at the same allocation site by a single abstract symbolic heap object. This paper begins similarly; Our analysis already differs from theirs by merely takeing note of the 'type' of the object allocated during program execution. Nevertheless, their description of "call site context sensitivity" seems to come close to how we describe our analysis. There are however significant differences between our analyses: they analyse Java Programs whereas we analyse Java Libraries; their primary interest seems to be in merely enumerating the distinct reachable calling contexts starting from the program entry point, whereas our analysis uniquely represents every called context reachable from every public entry point into the library; Their description of the actual call-graph is very brief, whereas, our thesis makes a effort to describe the invocation graph, and its construction with many examples.

The Analysis by Mayur Naik, *et al*: This work reports an analysis that improves upon the approach by Amy *et al* by incorporating the contribution of other related properties to improve the precision of the reports, in terms of minimising false positives. They describe, for closed Java programs, a k-object-sensitive-analysis that starts with k=1 for all heap objects. At this stage their heap is as large as the heap fetched for the analysis described by [8]. The analysis then re-iterates, assigning higher values of k for some objects causing the corresponding *symbolic heap object* to split into multiple objects, and so on, resulting in improvement in the precision of the analysis. Their analysis identifies 6 properties: all of which can be established or approximated using static analysis of the input. So identified basic facts regarding the program are expressed in the Datalog notation that is used by the `bddbddb` solver to arrive at the results of the analysis.

Using this approach they are able to identify deadlock possibilities in the program that include fewer false positives. This approach gives an excellent framework for deadlock detection in closed programs. Our analysis is suitable for libraries where there usually is a set of entry points. It would be interesting to use some of the ideas from [10] to develop programs to detect deadlocks starting from the lock order violations predicted by our analysis.

The analysis by Jyotirmoy, *et al*: This analysis is intended for libraries. They fetch a lock graph quite like [8], but they then prune it using type information and other ideas. They report significant reduction in the size of the graph. Using this pruned model, they then use SMT solvers to enumerate, eliminate, and thereby identify "patterns of aliasing between the parameters of concurrent library methods that may lead to deadlock". Their results list the analysis for quite a few APIs from the JRE, for all of which their analysis requires less than an hour. We observe the following:

- Since they do not analyse all the APIs together, in a single analysis, their approach

may not be able to find the three realisable lock order violations form our Table TC7.3, which involve types from two or three different APIs. Separately, however, they are able to find 3 deadlocks in `java.util`, and no deadlocks in `java.rmi` or `java.io` APIs.

- Their analysis would not find the 7 CORBA API violations from our Table TC7.3, since they all involve the use of `static synchronized` methods, and objects of the type `java.lang.Class` which they prune away in one of their pruning steps (see (c) under their Section 5).

- We have reported a bug in the Security API of Java 1.6 (Row 1 of our Table TC7.3) that drew some quick attention from the Java sustaining team in Oracle, and the violation is now fixed. In 2009, however, they report no deadlocks in the `javax.security`, & `java.security` APIs, in their Table 2.

- Similarly, the 3 deadlocks from the Sound/MIDI API from our Table TC7.3 are also not found by them. They report no deadlocks in the `javax.sound` API.

For all the bugs they refer to in their Table 3, our analysis does report lock order violations corresponding to all of them. They have not found the deadlocks from the MIDI and the Security APIs possibly because *either* they have analysed only the classes from the specific name spaces ignoring utility classes developed for that API, *or* some pruning resulting from their choice: "... In the presence of ... the fixpoint may not terminate. We ensure termination by artificially bounding the size of the access expressions considered by our approach." (end of their Section 2).

2.2 Other Interesting Approaches

Apart from approaches based upon static analysis, there are many other interesting reports in the literature relating to the problem of deadlocks in Java. Below we briefly

discuss some of the notable ones.

Fancong Zeng's work: [17] is an early work on this problem that explores the possibility of using exceptions as a way to break a deadlock after having reached it during normal program execution. It suggests the use of a TIMEOUT based mechanism to detect if a thread is deadlocked, and for the JVM to arrange for a runtime exception to be thrown onto its thread stack as a way for it to be brought out of the deadlock.

Subsequently, another report [18] prototypes the use of *Ghost Locks* to avoid reachable deadlocks during program execution. In this approach the JVM maintains a lock access order graph (LAOG). For any strongly connected components (SCC) that it may detect in the LAOG, it creates a ghost (gate) lock to protect any subsequent grants of locks from the corresponding SCC. This strategy is applicable to program use-cases where the same deadlock opportunities keep getting triggered repeatedly. The paper observes that, since building the LAOG and detecting the SCC in it requires the program to execute for a while, any deadlocks reached during that initialisation phase cannot be avoided by this strategy.

Another report [13] discusses pattern driven deadlock avoidance based upon the Just-in-time (JIT) compiler capability that is available in the JVM. In this approach the deadlock pattern must be first discovered, so that its occurrence may be avoided. A Few approaches to detect deadlock patterns, and some ways whereby their occurrence may be avoided are discussed.

Dimmunix: [12] describes a working prototype of a tool that actively prevents known deadlocks from manifesting during subsequent executions. It must witness a program execution that leads to deadlock. In the process it records the *deadlock fingerprint* which is read and used in subsequent executions to avoid that deadlock. The dynamic aspects of the deadlocked state are abstracted away when recording the fingerprint, so that it may be used (as is) during subsequent executions.

The Java prototype of Dimmunix comprises of a Java agent, and its deadlock avoidance strategy (*das*). The Java agent installs a bytecode transformer which employs the instrumentation API to engineer bytecode of classes loaded during application execution, inserting calls to *das* at appropriate points: before and after every `monitorenter` and also before every `monitorexit`. The *das* maintains a Resource Allocation Graph (RAG) whose nodes represent locks & threads. Its directed edges represent the acquired / requested relations amongst them. When program execution reaches an unknown deadlock, the RAG is used to construct its deadlock finger- print that is recorded. Subsequent executions read the fingerprints, and based on how the RAG evolves, *das* may block the acquisition of an (otherwise) available lock by some thread, to avoid a known deadlock.

M.Samak & M. Ramanathan's work: Two recent reports by them are based upon the dynamic analysis approach. Since they must start with observing program executions, their methods apply to whole programs.

[19] reports an interesting organisation of tools to help identify program executions that cannot deadlock from those that might, from a collection of program executions that other methods (for instance, those based upon static analysis) would identify as 'potentially deadlocking'. Their method uses instrumented program executions to generate traces for downstream analysis. Based upon such analysis they 'prune' away 'false positives' that cannot deadlock. The ones that can actually deadlock, they are able to 'replay' them so the execution actually deadlocks. Such tooling is useful since many times even when reachable deadlocks do exist, realising them during program execution is difficult.

[20] synthesises deadlock programs using dynamic analysis of sequential Java program executions. Execution traces of sequential single-thread test programs are recorded. Along such execution threads, lock dependency relations and lock order-

ings are noted. A program synthesiser then investigates these recordings and generates concurrent test programs that may share objects that are locked in opposite order in its two component threads stacks. Execution of such concurrent programs can therefore lead to deadlock. An advantage of this approach is that all the test execution traces that are used as a start point are known to be realisable. So, indeed, if the synthesiser does manage to generate a concurrent program, its execution is far more likely to deadlock than if the input were stack traces produced from static analysis like [8]. On the flip side, the effectiveness of this approach is critically dependent upon the coverage of the test-suite.

Pallavi, *et al* [21] reports a 2-stage dynamic analysis approach where first, the application execution is observed to fetch a trace program that models the execution. In the second stage, this trace program is fed into the JavaPathFinder (JPF) [22] model checker to explore all its possible inter-leavings, for deadlocks. They report significant reduction in the time required by JPF when fed the trace program as against the original program.

2.3 Our Goal is to Locate Lock Order Violations

The purpose of our analysis is to draw the programmers' attention to such places in the source code where locks are taken either in violation of the lock ordering principle or, (in any case) ignoring it.

Amy's report [8] describes a formulation for the deadlock-ability problem based upon an alias analysis of the source. Their work also demonstrated that for even medium sized APIs such an approach fetches model-sizes that tend, very quickly, to become intractable. [10], given a closed application, investigates whether its execution may create and share objects in a manner that its application threads may deadlock. [9] investigates libraries to identify aliasing relations amongst application objects, which

when used to invoke the library concurrently may cause deadlock.

We subscribe to the view is that if two types are locked in opposite order in different places in the source code, then (irrespective of whether their executions may/may-not share objects,) that itself is good enough reason to review their design and the use-cases. Bundling this view / concern conveniently in the form of a component in the language tool chain would help sensitise designers to this problem and enormously help contain its incidence in the code developed. In order for that to be feasible, the design and implementation of the analysis must be able to scale to the size of the typical large libraries.

Our observation (from the literature survey) is that for a code-base as large as the JRE, the alias analysis approach fetches a very large model, rendering the analysis intractable. The global OO type based analysis we describe in this thesis has not been reported before.

CHAPTER 3

OVERVIEW OF THE WORK

Contents: This chapter gives an overview of the thesis. In Section 3.1 we describe an approximate procedural analysis of the JRE that uses a call graph to capture reachability amongst method definitions. Section 3.2 motivates the need to distinguish between the definition and behaviour of methods, when analysing Java. Section 3.3 describes an object oriented analysis that improves upon section 3.1 by taking note of method behaviour as well as statement level synchronisation. Finally, section 3.4 briefly outlines the use and application of the results of such an analysis.

The mainstay of our work comprises of the design and prototype implementation of an analysis of Java with a view to identify such sets of entry points into the JRE that (each) violate the lock ordering principle. We have used, as a case study, version 1.6 of the JRE as representative of large libraries, to demonstrate the scalability of our approach. To demonstrate its usefulness, from among the various lock order violations reported therefrom, in 15 different cases, we have developed simple Java programs whose executions indeed deadlock as predicted. As another application of lock order violations reported by our analysis, we establish their ability to enhance the usability and effectiveness of deadlock avoidance using *Dimmunix* [12].

In general, our analysis starts by using `jar` files as input. For the JRE case study, we

input all the JRE-`jar`(s) generated by a successful build. From the input, the analysis discovers all the defined Classes and Interfaces, and the apparent caller-called relations between their method members. If the input were to represent code expressed in a procedural language, (like `C`, for instance), then a call graph could be used to represent such information. In fact, for the (exploratory) initial prototype of our analysis, we used a call graph to represent the caller-called relations as apparent in the input. The downstream analysis itself was, therefore, unsound and incomplete, but it nevertheless helped to detect one genuine deadlock in the Annotation API of version 1.5 of the JRE. We called the initial prototype analysis "Approximate Procedural analysis of Object Oriented Libraries" [23], more completely described, with a running example, in the section 3.1.

The analysis outlined in section 3.3 is an Object Oriented analysis of the JRE. It is not "Approximate" since it accounts for the programmers' use of `synchronized` statements, and it is OO (as against Procedural) since it handles Polymorphism, exception handling, and dynamic dispatch. It starts by using an *Invocation Graph* to record the information discovered from the input `jars`. Whereas a call graph records the apparent caller-called relations amongst defined methods, the *invocation graph* records the *invocation*s amongst *behaviour*s of invoked methods. Section 3.2 elaborates upon the distinction between the *definition* of a method, and its *behaviour*.

3.1 Approximate Procedural analysis of the JRE

The initial prototype [1] implements the analysis described in this section. For the complete discussion including the results fetched therefrom, refer [23].

Conservative Static Analysis of Java libraries to detect and report all *apparent* sources of potential deadlock necessarily reports a larger set of instances than actually *feasible*.

[1] https://github.com/vivekShanbhag/Stalemate/ApproximateProceduralAnalysis

The practical usefulness of such analysis depends upon the extent of *over-reporting* [24] (reporting of *infeasible* deadlocks). Clearly, the fewer such false reports, the better. *Under-reporting* is when some actually *feasible* deadlocks go undetected by an analysis. Even in such (incomplete, or approximate) analysis it is not the case that all reports are *feasible*: an analysis can therefore do both: include some infeasible deadlocks, while also miss out some feasible ones from its report. The analysis described in this section is such an analysis. More specifically:

The analysis is approximate since it discounts the use of `synchronized` statements, taking into account only the `synchronized` methods. It creates a single call graph to represent all the input, using information in the method headers to discover `synchronized` methods. A node of the call graph uniquely represents a corresponding method definition, and directed edges between nodes denote possible apparent caller-called relations. A directed path from node X to node Y implies a possible invocation of the method Y from code that is either part of the method X, or is invoked from its execution.

It is a *procedural* analysis because it discounts the dynamic dispatch feature (runtime type based method binding) of the Java Language.

Only 2-lock, 2-thread deadlocks reported: In its reporting step, only cycles involving 2 locks and 2 threads are reported.

Ignoring `native` methods: The investigation does not look for/at implementations of `native` methods declared in the JRE. This amounts to making a simplifying assumption: that native methods neither acquire any locks nor do they callback into the API.

Data is conservatively abstracted away: It being a static analysis, variable values can not be tracked. The analysis is therefore 'conservative': if there appears an invocation of method T from the bytecode for method C, then the possibility that T could be called from within C is represented by including the edge '$C \rightarrow T$' in the call graph.

The analysis comprises of steps 0 through 4, as detailed below:

Step 0: Specifying input: The input is the collection of jar files (under the `jre` subdirectory) produced by a successful build. Their successful build implies that the source is well formed; our implementation, therefore, need not be robust in the face of ill-formed classes, nor those from bad Java. A running example illustrates the function of every step in the algorithm. Consider the code below in which Classes `C`, `D`, and `E` represent library defined classes; and some class `F`, whose certain method members are referenced, but which is not defined by the library.

```
public class C  {
  public synchronized void l()    { E.m();  };
  public              void k()    { F.kk(); };
  public synchronized void a(D d) { d.b();  };
  public synchronized void z()    { F.zb(); };
}
public class D  {
  public static synchronized void n()   { F.p(); };
  public        synchronized void b()   { F.q(); };
  public static synchronized void x(E e, C c) { e.y(c);}
}
public class E  {
  public static void m()    { D.n(); };
  public        void y(C c) { c.z(); };
}
```

Step 1: Single pass call graph construction: Iterating over all methods of all classes in the input jars, the following substeps are executed for each method:

1.a: Create a node of the call graph: Create a node corresponding to the method definition. Store its fully qualified method name and its signature, as the key. Insert this *defined* node into a (keyed) list: `definedMethods`. When creating the node, if it is found in another list of `undefinedMethods`, then fetch it from there and populate its attributes before moving it into the `definedMethods` list. Continue to parse the full definition of the method header and body (Steps 1.b, 1.c). The analysis maintains two hash Maps: `undefinedMethods`, and `definedMethods`.

1.b: Tag the node if it is either `synchronized`, or `static synchronized`. Additionally, add all such tagged nodes into a set called `smNodes`. This set forms a subgraph

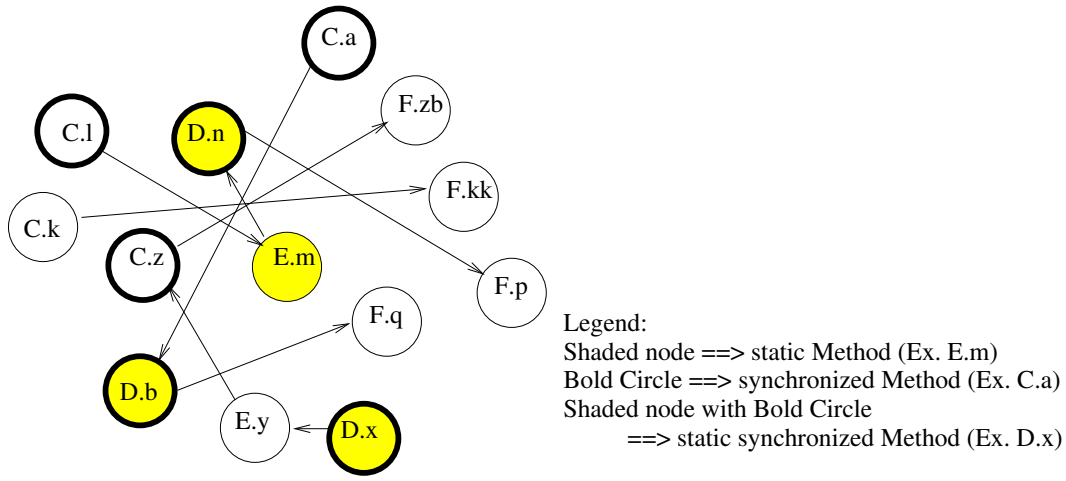

Figure FC3.1: The call graph constructed after Step 1.

of the entire graph, corresponding to only `synchronized` methods.

1.c: Populate its called methods attribute, `calledMethods` (of type `Set`): Parse through the bytecode corresponding to the method body, and for every method invocation (conditional, or otherwise), insert a directed edge between the current node, and the node corresponding to the called method. If there is no node corresponding to the called method, then create it like in Step 1.a, but insert it into the list of `undefinedMethods`, since it is still *not defined* (its method body hasn't been parsed, yet).

The step 1, above, takes a procedural view of the library methods, ignoring polymorphism, and constructing a procedural call graph. It is easy to see why this computation should terminate: it visits every class in every jar, exactly once, in a finite input. Figure FC3.1 illustrates the call graph after step 1, corresponding to code fragment in Step 0. The bold nodes correspond to `synchronized` methods, and the shaded ones indicate `static` methods. A directed edge indicates that the to-node *could be* invoked during the execution of the from-node. At the end of Step 1, nodes corresponding to methods of Class `F`, namely, `F.kk`, `F.zb`, `F.p` and `F.q` remain members of the `undefinedMethods`. Interestingly, although the manner in which they are invoked in the code indicates that they are `static` methods, however in Figure FC3.1 they are not 'shaded'. This is be-

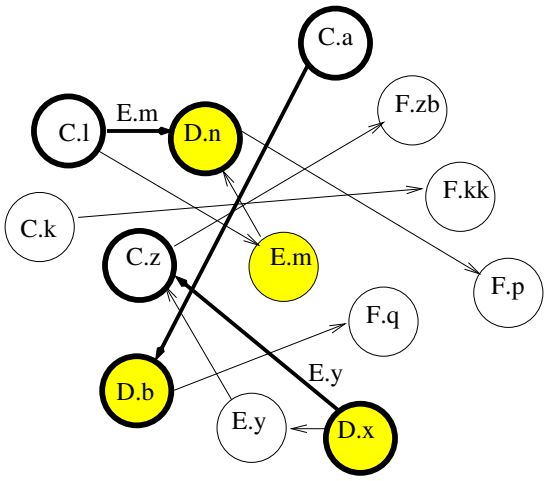

Figure FC3.2: The call graph after executing Step 2.

cause their definition is not seen in the input; It is only when the definition of a method body is parsed that the attributes of the corresponding node are populated.

Step 2: Discover, and record *reachability* amongst nodes associated with `synchronized` methods. Iterate over members of the `smNodes` set, populating their `smEdges` attribute (type: `Set`) as follows: insert a directed edge between two members of this set corresponding to a directed path from one to the other. Additionally, annotate the newly added edge with an ordered, hyphenated, concatenation of the names of nodes along the corresponding path between them. This edge annotation reflects the call stack that represents a *direct or eventual* potential call, with possibly other non-`synchronized` method calls in the call stack, in between the stack frames corresponding to the from-method, and the to-method.

If the cardinality of the set `smNodes` is, say, n then, in Step 2, we may create at most $n*(n-1)$ edges. Figure FC3.2 illustrates the graph after execution of step 2. The bold edges correspond to the newly added `smEdges`.

Step 3: Unconditionally drop nodes and edges corresponding to non-`synchronized` methods: we delete from the graph, all nodes, (and edges between them), that do not represent `synchronized`, nor `static synchronized` methods. Beyond this step in

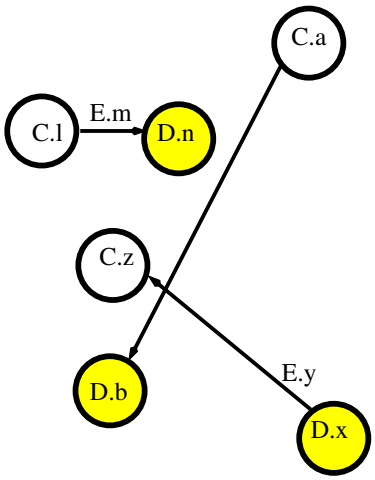

Figure FC3.3: The call graph after executing Step 3.

our algorithm, we are interested only in members of `smNodes`, and `smEdges` amongst them.

Each node in the set (`smNodes`) corresponds to the act of first acquiring a lock (or having to block on it), then doing some (useful) work, and finally releasing the lock. The directed edges `smEdges` represent the *possibility* of a thread trying to acquire *another* lock, while already holding on to the ones acquired before. Of course, the lock could be either on an *object* (instance lock), or on a *class* (type lock). Figure FC3.3 illustrates the state of the call graph after executing Step 3.

Step 4: Construct the *tlog*, and report Cycles: Nodes from `smNodes` represent methods, whereas we want to discover and report cycles on the *locks* that these methods contend for. Use the information in $< smNodes, smEdges >$ to construct another graph, called the *Type Lock Order Graph (tlog)*, and report cycles in it.

Corresponding to every *type* in the library under analysis, the *tlog* can have between 0 and 2 nodes. The three cases corresponding to a type called (say) `tN` are: if it has no `synchronized` methods, then the *tlog* has no node (0 nodes) corresponding to it; if it defines both (some) `synchronized` as well as (some) `static synchronized` methods, then the *tlog* has 2 nodes corresponding to it: one called *tN*, and another called

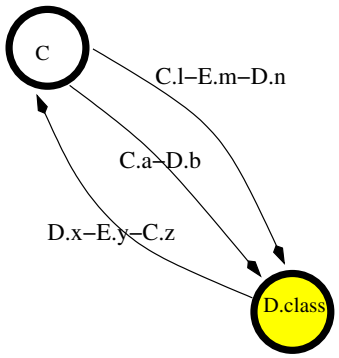

Figure FC3.4: The 'type lock order graph' (*tlog*) after executing Step 4.

tN.class; finally if `tN` defines either only (some) `synchronized` method(s), or only (some) `static synchronized` method(s), then the *tlog* has only one node to represent it, (respectively) called either *tN*, or *tN.class*.

For every member of `smEdges` in the call graph, we insert a corresponding edge in the *tlog*. The edge annotation on the original `smEdge` that represents the corresponding potential call stack is copied as the annotation on the *tlog*-edge.

A node *tN.class* in the *tlog* represents the *type wide* lock associated with type `tN`. A node *tN* in the *tlog* represents *collectively* all the objects of type `tN`. Each directed edge in the *tlog* represents a series of methods, (starting from a method of the class associated with the *from-node*), one calling into the next, that ends in a method of the class associated with the *to-node*. This graph has two types of nodes: those that represent instance locks, and others that represent type-wide-locks. They correspond to locks required for `synchronized` methods, and `static synchronized` methods, respectively. Finally, we report cycles in this graph.

Figure FC3.4 illustrates the *tlog*, as constructed by Step 4 from the graph in Figure FC3.3. For this example, the algorithm would print an XML format report as reproduced below.

```
<Cycle-2 C D.class>
```

```
    <Thread-1-Option>C.l E.m D.n</Thread-1-Option>
    <Thread-1-Option>C.a D.b</Thread-1-Option>
    <Thread-2-Option>D.x E.y C.z</Thread-2-Option>
</Cycle-2>
```

This report renders the information in the *tlog* into text output. Notice the tags: `Thread-1-Option`, and `Thread-2-Option`. They indicate that if there were to be two threads in an application program, one of which were to realise any of the stack traces under the 'Thread-1 Options', and the other were to realise any of those under 'Thread-2 Options' *concurrently*, then the lock order violation *may* lead to deadlock. In this example, the lock `C` is an instance lock, whereas the lock `D.class` is a type-wide lock. We refer to such XML output as a single *lock order violation* report, irrespective of how many thread stack options it contains.

3.2 Method Definition *vs* Method Behaviour

An important motivator for static analysis methods is that the size of such analysis is implied by the size of the program, rather than the size of the data range that the program operates upon. In languages like Pascal or C, the basic types are few, and there are just two language constructs that allow creating 'higher level' types: *struct*/*typedef* and *union*. Their 'type system' part of the language constructs has very little implication on the interpretation of program behaviour (over and above) as expressed through using the rest of the language constructs. Therefore, when static analysing a C program, the interpretation of the behaviour of the program, the size of (the expression of the) program, and the size of the analysis is roughly in direct proportion to each other.

Whereas, the same is not true with OO languages like Java/C++/C#. Comparatively, these languages have a much richer component intended for use to create 'higher order' types, and very interesting, and complex relations between them. Using such features of these OO languages, it is possible to express program behaviour in a more com-

pact form. Inheritance (in Java, for instance) is a very good example: all subclasses of `java.lang.Object` inherit all the public functionality programmed for it by the library. Thereby, every object in the application is able to (potentially) demonstrate the behaviour it so inherits at virtually zero cost, as reflected in the size of the (expression) of the application program. Such notational convenience can be used very effectively to create relatively brief specifications of program behaviour.

Therefore, when static analysis is conducted of such OO input, without giving due respect to the implications of its underlying type system, inaccuracy is bound to creep into the results. The description, in Section 3.1, of procedural analysis with no regard to the 'virtual function mechanism', is indeed inaccurate, but the size of that analysis was (roughly speaking) in direct proportion to the size of the (expression of the) input.

However, to conduct an OO analysis, one needs to *un-fold* the expression of the input with due respect to its underlying type system, to discover the complete programmer intended interpretation of program behaviour. Such analysis therefore needs to create a representation of the program that corrects for the notational convenience provided in the form of the type system constructs. Thus, while the size of such analysis is still in direct proportion to the program size, the constants are larger than that for an imprecise analysis.

For instance, consider the Java code, below, that defines an interface called `DoBad`, two classes `DivByZero` & `LoopForever` that implement it, and an `Observer` class that provides the `public static main`. The `javac` compiler generates a single method body for the definition of `Observer.today()`. However its two behaviours (as intended by the programmer) in its two invocations in the `main` method, are quite different. This is because they receive invocation parameters of different runtime types causing the executing JVM to dynamically bind the two invocations differently.

`"Observer.java"` ≡

```
interface DoBad { public void doIt(); };
class DivByZero implements DoBad {
  public void doIt () { int retval = 55/0; };
};
class LoopForever implements DoBad {
  public void doIt () { for ( ; ; ); };
};
public class Observer {
  synchronized static void today (DoBad howTo) {
    howTo.doIt();
  };
  public static void main (String[] args) {
    today (new LoopForever());
    today (new DivByZero());
  };
}
```

To provide for the need to represent the implications of such dynamic dispatch, we introduce a graph called the *Invocation Graph*. Whereas the conventional call graph uses nodes to represent merely method definitions, our invocation graph contains *additional* nodes to represent every such method behaviour fetched from the dynamic dispatch of methods based upon the runtime types of objects involved at the call site.

Therefore, although the input source contains only one definition for the method called `Observer.today()`, our OO analysis creates as many distinct models of its various behaviours as can manifest owing to the runtime method binding feature of the language. The analysis views methods defined in the input as being template specifications for method behaviour, and instantiates as many of its *variant* behaviours as can manifest, given the different combinations of runtime types for input parameters. All these instantiated variant behaviours are recorded as nodes of the invocation graph, and referenced wherever the lists of actual parameter types match precisely.

The example above demonstrates one source of variability in the behaviour of a defined method: that which is sourced from the possibility of invoking it by passing parameters of runtime types subclassed from those specified as the declared types of the corresponding arguments. Another source of variability in the behaviour of a defined method results from the possibility of invoking an inherited method on an object

subclassed from the type that defines the method.

3.3 OO Analysis of the JRE

In this section we describe an evolved form of the analysis: 'Accurate OO Analysis of the JRE'. More specifically, it takes cognisance of both method level, as well as statement level use of `synchronized`, accounts for dynamic dispatch, and reports *upto* n-cycle lock order violations, for configurable values of n. The analysis is conducted by a single execution of the tool: it reads all the input, computes the analysis and reports uniquely the violations of lock ordering. Its 4 high level steps are briefly described below. (The invocation graph, abstract interpretation, and the type lock order graph are detailed in chapters 4, 5 and 6, respectively).

Step 1: Initialising the Invocation Graph: For our case study, it reads in all the classes and interfaces from all the *jar*s of the JRE to start building an *invocation graph* that represents it. Initially, nodes of this graph are created corresponding to every method definition fetched in the input. All these nodes correspond to method definitions, and the bytecode that defines the corresponding method body is retained as their attribute. Initialisation of the invocation graph in this manner is complete once all the input has been exhausted. The so initialised invocation graph contains only nodes, and no edges. However, subsequently the invocation graph evolves as its nodes are subject to *abstract interpretation*. The evolution adds directed edges, and also adds more nodes. The invocation graph extends the traditional call graph structure, in two ways (see illustrative example in section 4.2, figures FC4.1, and FC4.2):

- `synchronized` **statements**: apart from having nodes to represent entire methods, it can have nodes to represent `synchronized` statements.

- **Behaviour of code**: In addition to having nodes to represent the *definition* of entire methods or `synchronized` parts thereof, it has nodes to represent different (*variant*) behaviours fetched from invoking methods in different calling contexts, as discussed in the Section 3.2.

A typical compiler execution limits its view to the input files, and the structure of the library classes referenced therefrom. Moreover, it propagates only the declared types of variables and references rather than the actual types of the objects assigned into them. Our static analysis differs from the typical compiler on both these accounts. It takes a global view of all the classes defined in the JRE, and when analysing a specific method invocation, it chooses to propagate information regarding the assigned types (of references / variables) rather than their declared types. Consequently the invocation graph built by our static analysis attempts to model dynamic behaviour arising from the following sources:

- Invoking the method defined by a superclass on an object of a subclass.
- Invoking a method by supplying it objects of a subclass as arguments where the defined parameter type is that of the super class.
- Invoking an interface method on an object of a type that implements it, or its subtype.

In the general case, this may not[2] be always possible owing to the rich expressiveness of the Java Language given its various APIs (including The Reflection API). However, we find that for bulk of the classes and interfaces defined in the JRE, it works quite well.

Step 2: Discovering the complete Invocation Graph:

[2]Section 9.1 presents an example of a conforming Java program whose execution can deadlock (in two different ways); But, its static analysis, however smart and advanced, may never reveal such a possibility.

The intent of this step is to discover the various behaviours (of methods) that are reachable from the various entry points into the library. The JRE, for instance, defines 61,500 public concrete[3] entry points. Rooted at each of these, we discover the set of reachable method behaviours that may be reached on the thread stack that invokes the root method. This is achieved by *collecting & propagating* information regarding the *runtime types* of objects, where available. The type information is propagated across method boundaries, making it an inter-procedural analysis. This computation is conducted one method at a time, in an iterative manner. We refer to this computation as the *Abstract interpretation* of the method body. During such abstract interpretation, (any) synchronized portions of the method bodies are identified, and represented by separate nodes in the invocation graph.

During abstract interpretation the calling context for method invocation is accurately estimated facilitating correct identification of the dispatched method. As a result, new nodes to represent variant behaviour are added to the invocation graph. The so discovered behaviours in different calling contexts are stored uniquely in the invocation graph, thereby ensuring that this computation terminates.

Every path rooted at a node corresponding to a `public` method of a `public` class in the invocation graph, above, represents a possible thread stack configuration that *may* be reached by some application thread invoking into the library from that entry point. Whenever multiple locks are acquired along such a path, it implies an ordering among the objects *or* their respective types.

Step 3: Inferring the *type lock order graph*: The analysis, next, constructs a *type lock order graph (tlog)* (quite similar to the one described in section 3.1) to collectively represent orderings amongst types as implied by all such paths in the invocation graph along which multiple locks are acquired. The nodes in the *tlog* correspond to the types

[3] A method is concrete if the type of its target object and the types of all its formal parameters are classes whose objects may be instantiated.

input to the analysis. Directed edges amongst them imply ordering as reflected by the corresponding thread stack configuration. Cycles in the tlog therefore correspond to sets of *reachable* thread stack configurations which together may violate lock ordering amongst the types represented by the corresponding nodes.

Step 4: Reporting lock order violations: The analysis reports all cycles in the tlog involving upto n nodes, where n is configurable (default value: 4). For a k-subset of types $T_1, T_2, ...T_K$, all k-cycles involving objects of these types are reported together. It contains the following information in XML format: $\langle T_1, ...T_i, ...T_k \rangle \equiv Ts_1, ...Ts_i, ...Ts_k$, where $T_i, \forall 1..i..k$ are names of nodes of the tlog. In general, Ts_i lists the edge annotations of all directed edges $\langle T_i, T_{i+1} \rangle$. Finally, Ts_k lists the edge annotations of all directed edges $\langle T_k, T_1 \rangle$.

3.4 Results of the Analysis and Applications

Ideally, an analysis like the above ought to be performed by a component of the Java tool chain. Such a component can be used during regular cycles of code integration to compose an API, and composition of APIs to form a library. Also when making an update release that may contain few tens of bug fixes, to ensure that it does not accidentally add any lock order cycles.

Developing deadlock programs, given lock order violations: We demonstrate the usefulness of the analysis to predict existing deadlocks in the JRE. From the many lock order violations in JRE 1.6 reported by Stalemate, we identified a few involving 2 and 3 lock types, for further investigation. After studying the corresponding source of the JRE we developed, in 15 cases, simple programs that realise the corresponding lock order violation, resulting in deadlock. Two of these are on a set of three locks, involving three threads. One 2-cycle deadlock was in the security API that has now been fixed. Many thread stacks from the 15 programs acquire one of the locks using synchronized

statements within the method body. An interesting thread stack, that participates in 3 deadlocks, contains a dynamic method dispatch which could not have been predicted using conventional static analysis based upon declared types. A detailed discussion of the deadlocks is in chapter 7; we have filed bugs for all the above deadlocks (See Table TC7.3).

Enhancing the usability and effectiveness of Dimmunix: Our analysis can be used to enhance the usability and effectiveness of a tool for deadlock avoidance called *Dimmunix* [12]. Dimmunix operates in two modes: when it witnesses some deadlocked execution for the first time, it records a *fingerprint* of the deadlock for future reference; whenever a subsequent execution of the same program *seems* to likely enter into some such already witnessed deadlock, Dimmunix avoids it from similarly deadlocking. We demonstrated [25] that the information available in the lock order violations reported by our analysis can be rewritten as deadlock *fingerprints* for Dimmunix so that it need not, anymore, witness deadlocks so as to avoid them. Chapter 8 has the details.

Summary: This chapter gave an overview of the thesis. It started by describing a procedural analysis of the JRE using a call graph to capture the input. Arguing the need to distinguish between the definition and behaviour of Java methods, it then briefly introduced an invocation graph that can be used to represent definitions as well as behaviours of Java code fragments. This was followed by an overview of the organisation of an object oriented analysis of the JRE. The chapter closes with a brief outline of the use and application of the results of such analysis.

The next chapter describes the invocation graph in greater detail.

CHAPTER 4

THE INVOCATION GRAPH

Contents: This chapter presents a detailed discussion of the Invocation graph. Section 4.1 introduces the *Invocation Signature* used to capture the calling context at an invocation site, and compares it with the *Method Signature* used to identify method definitions. Section 4.2 discusses the composition of the invocation graph, including the semantics associated with its different types of nodes and edges. Section 4.3 argues, using actual numbers for JRE 1.6, that computing the complete Invocation graph may not be feasible within the limits of a desktop computer. We identify a managable superset of the nodes that represnt all defined methods which includes all referenced behaviours to fetch useful insights for the library developer. This chapter is a rigorous treatment of the structure and organisation of the graph created by Steps 1 and 2 of Section 3.3.

The invocation graph is designed to be able to represent all *reachable* behaviour of code that is available in the 'input to the analysis'. In general, its nodes represent definitions and behaviours of code fragments from the input. In particular, all nodes created during its initialisation represent definitions of entire method bodies. Subsequently, after exhausting all the input, the rest of the nodes are identified, created through *abstract interpretation* of the existing nodes. This continues until all reachable behaviours are

subject to abstract interpretation.

Chapter 5 discusses *abstract interpretation* of a method body in detail, but suffice here to say that its intent is to *identify* the runtime types of objects and to *propagate* them with precision to every call site, and synchronisation site. This allows the analysis to identify the precise *invocation signature* corresponding to every potential invocation from the method body that is subject to abstract interpretation.

4.1 Invocation Signature

The purpose of an Invocation signature is to capture information regarding the actual (runtime) types of all arguments and of the target object involved in an method invocation at a call site. For instance, consider the two invocations of method signature '`Observer.today:(LDoBad;)V`' in the `Observer.main(...)` method from `Observer.java` in Section 3.2. Notice that it is the same method that is invoked twice, but the argument passed to it in the two invocations are of different types. The compiler resolves the two calls to the same method based upon the compile time declared type of the argument. However since (we shall see in chapter 5, how) *abstract interpretations* of `Observer.main(...)`' collects and propagates information regarding the runtime types of objects involved in the two invocations, it is able to construct the invocation signatures corresponding to the two invocations as, respectively, `Observer.today:(LLoopForever;)V`, and
`Observer.today:(LDivByZero;)V`.

The bytecode for a method body contains the method signature of the compiler resolved target method, co-located with the members of the `invoke` family of instructions. Operationally, it is when such instructions are encountered during *abstract interpretation* of the method body, that the corresponding invocation signatures are composed to capture the runtime types of all the objects involved in the invocation.

Structurally the invocation signature is identical to a method signature. In case of an instance method invocation, its prefix is the fully qualified type name corresponding to the runtime type of the target object. Alternatively, in case of a static method invocation, its prefix identifies the fully qualified type name of the class that defines the target method. The unqualified method name and the cardinality of its parameter list are two pieces of information that the invocation signature inherits as-in from the corresponding target method signature generated by the compiler. The runtime types of the various invocation arguments are discovered during *abstract interpretation* and encoded into it, instead of the encoding of the formal parameter list.

4.1.1 Method signatures *vs* Invocation Signatures

Method signatures have a nice property: for well formed applications and libraries, they are unique. The same is not true for the invocation signatures. Although structurally they are identical, their functions are different. A Method signature is a mechanism to uniquely identify the corresponding method definition, whereas an invocation signature serves the purpose of collecting the runtime types of objects involved at a call site. Consider the following example which defines an interface called Y, a class M that implements Y, and class N that extends M.

"N.java" ≡

```
import java.util.*;
interface Y { public void yy (Collection c);  };
class M implements Y {
  public void yy (Collection c) {  System.out.println ("In M.yy");  };
};
public class N extends M {
  public void yy (SortedSet ss) {  System.out.println ("In N.yy");  };
  public static void main (String[] args) {
    SortedSet ss = new TreeSet ();
    Y  odYiN     = new N (); // object defined Y instatiated N
       odYiN .yy (ss);
   ((N)odYiN).yy (ss);
  };
};
```

The last two lines in the `main` method invokes method `yy` twice. The *invocation signatures* for both these invocations is the same (*i.e.* `N.yy:(LTreeSet;)V;` because the run time type of the target object (`odYiN`) is `N`, and that of the only method argument is `TreeSet`), however, (as witnessed from the relevant excerpt, below, of the text rendering (by '`javap`') of the corresponding '`N.class`'), they are resolved differently by the compiler (based upon the different declared types of the reference to the target object): the first one is resolved to interface method `Y.yy:(LCollection;)V`, while the second is resolved to method `N.yy:(LSortedSet;)V`.

```
'javac N.java; javac -c -classpath . N | less' fetches
...
public static void main(java.lang.String[]);
  Code:
   0:   new     #5; //class java/util/TreeSet
   3:   dup
   4:   invokespecial   #6; //Method java/util/TreeSet."<init>":()V
   7:   astore_1
   8:   new     #7; //class N
   11:  dup
   12:  invokespecial   #8; //Method "<init>":()V
   15:  astore_2
   16:  aload_2
   17:  aload_1
   18:  invokeinterface #9,  2; //InterfaceMethod Y.yy:(Ljava/util/Collection;)V
   23:  aload_2
   24:  checkcast       #7; //class N
   27:  aload_1
   28:  invokevirtual   #10; //Method yy:(Ljava/util/SortedSet;)V
   31:  return

...
```

Further, (as demonstrated by execution of N, as from:'`java N`') they are also dispatched differently by the JVM: the first invocation (*via:* invokeInterface) binds with method `M.yy:(LCollection;)V;`, while the second invocation (*via:* invokevirtual) binds with method `N.yy:(LSortedSet;)V;`.

Therefore, although the format for invocation signatures & method signatures are identical, our analysis needs to take care to ensure that signatures from the two sources

are not confused for one another. Further, since different behaviours may actually have identical invocation signatures, they must be further qualified appropriately to ensure that different behaviours do indeed have different names.

4.2 Composition of the Invocation Graph

During its initialisation, the nodes of the invocation graph are created in correspondence with every method definition encountered in the input. The methods may or may not be `synchronized`, and their method bodies may or may not include the use of any `synchronized` statements. The initialisation is complete once all the input is exhausted, and thereafter nodes are scheduled for investigation (using abstract interpretation) based upon criterion described in section 4.3. During abstract interpretation, the bytecode corresponding to the method body is used to discover the edges amongst existing nodes, as also new nodes to represent new behaviour. Additionally, as a result of abstract interpretation, nodes corresponding to methods which use `synchronized` statements get split up into multiple nodes as necessary. Section 5.4 discusses in greater detail how this is affected. Section 5.5 further illustrates this for the code in Figure FC4.1.

As mentioned earlier, nodes of the invocation graph represent definition or behaviour of either entire method bodies or `synchronized` portions of it, and directed edges amongst them signify invocation. Below, in greater detail, we describe the nodes, edges, and their correspondence to the definitions and behaviour of code fragments fetched in the input. An illustrative example: code in Figure FC4.1 defines classes A, B, and D, and Figure FC4.2 depicts the corresponding invocation graph.

- **Node to represent an entire method definition**: Defined methods that do not use `synchronized` statements are represented entirely by a single node in the graph.

```
class A {
  public int i;
  public void setI (int j) {
    i = j;
}};
```

```
class B extends A {
  public int ii;
};
```

```
public class D {
  public void gain(A a)  {
    synchronized (a) { a.setI(33);
      synchronized (this) { a.setI(77);
  }}};
  public void plain(boolean q)  {
    A a = q ? new A() : new B();
    gain (a);
}};
```

Figure FC4.1: Code from file: D.java

The method, itself, may or may not be `synchronized`. However, its corresponding bytecode does not contain any `monitor` instruction. The node name for such a node is the same as the corresponding fully qualified method signature. The *behaviour* of the corresponding method, when/where invoked exactly[1] as defined, is represented by this node: the invocation signature for such invocation is identical to the method signature.

The nodes `D.plain:(Z)V` and the three compiler generated constructors, namely, `A.init:()V`, `B.init:()V` and `D.init:()V` are examples of such nodes. The graph contains the node named `java.lang.Object...` since the compiler generated constructors contain calls to it.

- **Node to represent a *variant* behaviour of an entire method**: When the invocation signature (say *invSign*) is different from the compiler resolved method signature (say *methSign*), then based upon the semantics of the specific variant of the `invoke` instruction and in view of the type hierarchy, the analysis infers the target method (say *behSpec*) of the dynamic dispatch, as the JVM would perform it. During execution, upon method dispatch, the method body of *behSpec* is executed by passing it the arguments whose actual types are encoded into *invSign*. The resultant behaviour is represented by our analysis using the node, named: '*invSign-*

[1] which means to say that the runtime types of all its arguments, and also of the target object, are identical to the corresponding formal parameters, and the containing class, respectively, of the method definition

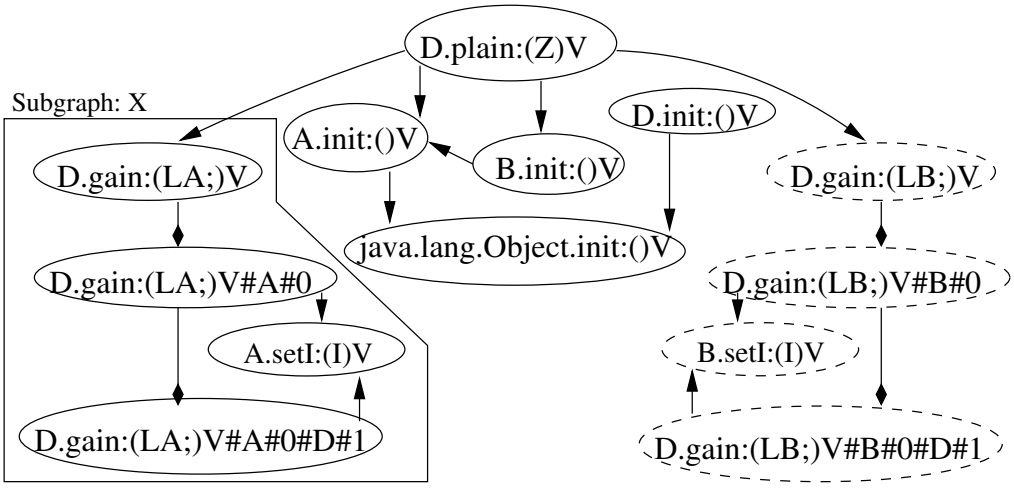

Figure FC4.2: Complete Invocation graph for classes: D, B, and A.

varientOf-*behSpec*-compiledAs-*methSign*'. When *behSpec* and *methSign* happen to be identical, the name is shortened to '*invSign*-varientOfAndcompiledAs-*methSign*'.

In Figure FC4.2, the node named `B.setI:(I)V` is such a node. For brevity only the *invSign* position of it is depicted in the picture. The complete name for the corresponding behaviour is `B.setI:(I)V-varientOfAndcompiledAs-A.setI:(I)V`.

- **Set of nodes to represent method definition / behaviour**: For methods that use `synchronized` statements, the invocation graph contains additional nodes that represent the `synchronized` parts of their behaviour. The root node represents the non `synchronized` code/behaviour fragments. The name of the root-node is chosen as per one of the above. The additional nodes form a tree structure rooted at the root node, and they are given unique names derived from a mangling of the name of the node corresponding to the enclosing code, and the type of the object locked by the corresponding code.

  As an example of such nodes, in Figure FC4.2, the set of nodes in the 'Subgraph: X' together comprise the representation for the body corresponding to method signature `D.gain:(LA;)V`. Notice that the root node of subgraph X has the same

name as the method signature. Only one node is reachable from the root node: that which corresponds to the first use of `synchronized` in the corresponding method body. Its name has three parts: the first being the name of the node corresponding to the enclosing code, next is the `typeName` of the object that is locked, and the last is an integer value to ensure uniqueness. Therefrom, two other nodes are reachable: one corresponds to the invocation of `a.setI()`, and another corresponds to the nested `synchronized` block. A similarly structured subgraph appears at the right hand side of the figure, corresponding to the behaviour of `D.gain()` invoked with a parameter of type `B`. Its root node's complete name, therefore, is actually `D.gain:(LB;)V-varientOfAndcompiledAs-D.gain:(LA;)V`.

- **Directed edges to represent Invocation**: A method invocation from a caller method, to a called method is represented by a directed edge correspondingly, from the caller node to the called node. Therefore, for a method invocation from a `synchronized` statement block, the edge originates from the corresponding non root node. Bulk of the edges in Figure FC4.2 are of this type. Notice that although `gain()` is invoked only once in the method `D.plain()`, since the parameter to its invocation may be of either type `A` or type `B`, there are two directed edges from node `D.plain:()V` to the corresponding behaviours to represent this possibility.

- **Directed edges to represent Containment**: From the root node of a method, to the node corresponding to a contained synchronized statement, we instantiate a directed edge, to signify containment, and potential natural control flow therefrom. Edges with a diamond arrow head in the fig. FC4.2 (pointing into nodes with the # symbol in their names) represent such edges.

Besides being named, the nodes of the invocation graph have another attribute (of type `String`): `lockType`. This attribute stores the type of the object to be locked, *iff* the corresponding method or code/behaviour fragment is `synchronized`. This attribute of

nodes representing non-`synchronized` code/behaviour is null.

In order to discover the invocation graph, as described above, apart from being able to precisely identify the scope of a `synchronized` usage, we also need to discover the type of the object locked by the corresponding `monitorenter` instruction. The content of the "Exception table:" associated with the method bytecode is used to precisely demarcate `synchronized` code fragments. The runtime types of the associated locks are discovered through abstract interpretation of the method body.

4.3 Complete Invocation Graph for a Library?

A typical API is a collection of co-operating Classes and Interfaces that can ease application development. An entire OO Library can be a collection of many such APIs. Our case study, the JRE Version 1.6, is such an OO Library that contains new and legacy Java code developed over a period of over 15 years. It contains over 200 different APIs, that together define over 16,500 Classes, 2,300 Interfaces, and thousands of 'inheritance' and 'implements' relations between them. A count of the concrete public methods offered by the entire JRE 1.6 is over 61,500. An attempt to create a *Complete Invocation Graph* for a library as large as the JRE using the desktop computer we have used for this research would have been quite futile.

Consider (for instance) some concrete public class, (say) `CPclass` from the JRE, and one of its public void instance method member called, (say) `pvMethod`, which requires no arguments. In an application program, `pvMethod` can be invoked upon an object of type `CPclass`, or of any of its subclasses. For every different concrete subclass of `CPclass`, a new node to represent the behaviour of `pvMethod` invoked upon a target object of that type needs to be created. Creating nodes for representing every such 'in principle' exported *behaviour* from the Library is prohibitively expensive. The JRE includes over 35,000 inheritance relations amongst its classes, and over 22,000 'Class

Table TC4.1: Class & Interface hierarchy rooted at Class java.util.TreeSet

Class / Interface Name	Defines/Declares[2]	Inherits	Implements
Class java.util.TreeSet	36 methods	java.util. AbstractSet	java.util.NavigableSet java.util.Cloneable java.io.Serializable
Class java.util.AbstractSet	4 methods	java.util. AbstractCollection	java.util.Set
Class java.util.AbstractCollection	16 methods	java.lang.Object	java.util.Collection
Class java.lang.Object	13 methods	–	–
Interface java.util.NavigableSet	18 methods	–	java.util.SortedSet
Interface java.util.SortedSet	6 methods	–	java.util.Set
Interface java.util.Set	15 methods	–	java.util.Collection
Interface java.util.Collection	15 methods	–	java.lang.Iterable
Interface java.lang.Iterable	1 method	–	–
Interface java.lang.Cloneable	1 method	–	–
Interface java.io.Serializable	1 method	–	–

implements Interface' relations. Therefore, while an analysis that requires such a large model can appear to be well *formulated*, it may not be completed as easily.

To better clarify the above, we use an illustrative example of an actual class from the JRE 1.6: '`java.util.TreeSet`'. Table TC4.1 has details regarding method counts of all its superclasses classes and all the interfaces it implements. For instance, the first row of the table reveals that the Class `TreeSet` *itself* defines 36 methods and inherits functionality defined for its superclass `AbstractSet`. It implements the three interfaces: `NavigableSet`, `Cloneable` and `Serializable`. An object of type`TreeSet` can be used to invoke any of the methods that either it defines itself, or that any of its superclasses define. The total number of such methods is, therefore, (36 + 4 + 16 + 13 =) 69.

A *complete* model of the library would have to model every one of these method dispatches for all classes in the Library, irrespective of whether the functionality is referenced or not. However, our modelling of behaviours in the JRE is need-driven. To begin with, we do model all defined methods on the types that define them. Therefore, in the above example, the 69 methods are all represented by individual nodes. However, only 36 of them have `TreeSet` as the type of the target object. 16 (of them) have

`AbstractCollection` as the type of their target object, and so on. (These 69 nodes are created during the Step 1 of the algorithm described in Section 3.3). Subsequently, during Step 2 of the analysis described in Section 3.3, it is discovered that only the following three methods that `TreeSet` inherits from its superclasses are referenced by the rest of the JRE. Consequently we explicitly model only these, individually, as nodes of the Invocation Graph.

- ```
  java.util.TreeSet.toArray:([Ljava/lang/Object;)[Ljava/lang/Object;
    -:varientOf:-
  java.util.AbstractCollection.toArray:([Ljava/lang/Object;)[Ljava/lang/Object;
  ```

- ```
  java.util.TreeSet.addAll:(Ljava/util/Collection;)Z
    -:varientOf:-  java.util.AbstractSet.addAll:(Ljava/util/Collection;)Z
  ```

- ```
  java.util.TreeSet.addAll:(Ljava/util/TreeSet;)Z
    -:varientOf:-  java.util.AbstractSet.addAll:(Ljava/util/Collection;)Z
  ```

The subsection below describes the construction of a superset of the initial Invocation graph, that includes nodes to model all such behaviours exported by the Library, that are *referenced* from elsewhere in its implementation. It is interesting to note that this approach facilitates an analysis that seems to be readily doable, and has fetched useful results for the JRE.

4.3.1 Discovering the rest of the Invocation Graph

Recollect that the invocation graph is initialised by reading in all the classes and interfaces that are input to the analysis. Next, a note is made of all the publicly accessible concrete entry points into the codebase. Subsequently, the transitive closure of all *behaviours* reachable from all such (public concrete) entry points is augmented into the invocation graph.

During the initialisation step, the parent-child relations expressed by the `extends` clause are taken note of, amongst both the classes and interfaces. Similarly, the relations
  ```

expressed by the `implements` clause are noted. Names of `abstract`, and `native` methods are maintained in separate lists, since their corresponding definitions are not available. All such *look-aside* information is necessary when conducting the analysis.

After exhausting the input, a single iteration over all the defined methods identifies all the *concrete* public entry points into the codebase. Starting from each one of these, *abstract interpretation* of the method body is conducted to identify all behaviour that is immediately reachable from there. This exploration is then repeated for all of *concrete* public entry points into the Library, and so forth, until no new behaviours are reachable anymore. Abstract interpretation is not conducted recursively. It is computed iteratively, one method body at a time: Once abstract interpretation for a node (say `caller`) is initiated, it is run to completion before initiating it for any other node, including any node (say) `called` by `caller`. In general, a node of the invocation graph needs to be subject to *abstract interpretation* only once. However, as we shall see in greater detail in the next chapter, if abstract interpretation uses the *precise but elaborate* approach instead of the *quick but imprecise* approach then a small number of nodes (1,383 nodes out of a total of 2,20,000 nodes) must be subject to abstract interpretations before other nodes that may have directed edges into one of these.

Summary: This chapter discussed structure of the Invocation graph. It defined the Invocation Signature used to capture the calling context at invocation sites and explained, using an example, how invocation signatures may not uniquely represent behaviour. The semantics associated with the different types of nodes and edges is discussed in detail using an illustrative example. Then, using actual numbers for JRE 1.6, the infeasibility of computing and representing the complete Invocation graph within the limits of a desktop computer is argued. A superset of the methods defined in JRE, which includes all method behaviours referenced by it, is described along with its representation.

The next chapter discusses ‘Abstract Interpretation’.

CHAPTER 5

ABSTRACT INTERPRETATION OF BYTECODE

Contents: This chapter presents, in detail (including illustrative examples in section 5.5), the abstract interpretation of a method body. With a view to (further) motivate its need, section 5.1 identifies the outcomes computed as results of abstract interpretation and their significance in connecting nodes of the invocation graph. Section 5.2 describes a stepwise procedure, using the contents of the 'Exception Table', to discover the offsets of the last `monitorexit` instructions that mark the closing of scope of every corresponding `synchronized` statement. This mapping is a necessary pre-requisite for abstract interpretations of the methods that employ `synchronized` statements. Section 5.3 discusses the *abstraction* and the initialisation aspects of abstract interpretation. The actual *interpretation*, involving details of the design of the transfer function are discussed in section 5.4. Section 5.5 illustrates the details of how abstract interpretation is carried out for methods in the illustrative example listed in Figure FC4.1. Section 5.6 contains more examples of the transfer function for some other interesting instructions. Finally, the discussion closes with a brief comparison of abstract interpretation and bytecode verification conducted by the JVM. This chapter presents a rigorous treatment of the analysis of individual method bodies that helps evolve the invocation graph in Step 2 in Section

3.3.

The intent of abstract interpretation is to collect and propagate information regarding the runtime types of objects. The reliability of the results of the analysis depends upon how accurately the collection and the propagation is achieved. The collection of the information is not much of a problem; Its systematic propagation offers some design options to choose from. Loss of information regarding runtime types of objects may occur during its propagation. One source wherefrom such loss of information may occur is while handling of the return type of a method. Another source is the handling of a read operation on either a class or instance attribute. In both cases there are two design options to choose from: (i) a lossy option that allows imprecision to creep into the results of the analysis, but is conveniently implementable quickly, (ii) another option that ensures loss less propagation of runtime information, but requires a more elaborate implementation that is more demanding on both memory and execution times.

Below, using an example, we illustrate code fragments that *may* lead to loss of information regarding runtime types of objects during its propagation, that can translate into imprecise results.

```
abstract class ObjectIn2D {
    protected       int sz = 0;
    public abstract int area();
    public abstract int perimeter();
};
class Square extends ObjectIn2D {
    public Square (int x)   { assert (x >= 0);  sz = x; }; // x: side of the square
    public int area ()      { return sz * sz;           };
    public int perimeter () { return sz + sz + sz + sz; };
};
class Circle extends ObjectIn2D {
    public Circle (int r)   { assert (r >= 0);  sz = r; }; // r: radius of the circle
    public int area ()      { return 22 * sz * sz / 7;  };
    public int perimeter () { return 22 * 2 * sz / 7;   };
};
class EqTriangle extends ObjectIn2D {
    public EqTriangle (int x)  { assert (x >= 0);  sz = x; }; // x: side of the eq.triangle.
    public int area ()         { return ((int)(java.lang.Math.sqrt(3.0) * sz * sz / 4.0)); };
    public int perimeter ()    { return sz + sz + sz;      };
```

```
};
public class Geom {
    static ObjectIn2D o = null;
    static public ObjectIn2D createObject (char type, int size) {
        switch (type) {
            case 'C' : return new Circle (size);
            case 'S' : return new Square (size);
            case 'T' : return new EqTriangle (size);
            default  : assert (false); return null;
        }
    }
    static public void main (String[] args) {
        if (args.length != 2) System.out.println (``Usage: java -cp . Geom [CST] [0..9]'');
        o = createObject (args[0].charAt(0), (new java.lang.Integer(args[1])).intValue());
        int      diff  =  o.area() - o.perimeter();
             if (diff > 0)  System.out.println (``Big.'');
        else if (diff < 0)  System.out.println (``small.'');
        else                System.out.println (``Beautiful.'');
    }
}
```

Consider the code in file `Goem.java` that defines an abstract class `ObjectIn2D`, concrete classes `Circle`, `Square` and `EqTriangle` that extend it, and a class `Geom` that uses these classes. Each of the three classes that extend `ObjectIn2D` define a constructor and two methods `area` and `perimeter`. The `Geom` class defines the '`static public void main()`' method that requires two invocation arguments: the first identifies the type of shape as being either **C**ircle, **S**quare, or **T**riangle, and the other argument is an integer value that specifies the **size** of the object. These two arguments are passed as arguments to its `createObject` method that creates the specified object. A reference to the object is stored in a class attribute `o`. Subsequently, based upon its area and perimeter, the object is judged as being either small (if its area is less than its perimeter), or Big (if its area is larger than its perimeter), or beautiful (if both are (roughly) equal). When analysing the `Geom.main()` method, notice the following:

- Modelling the type returned by method invocation: Notice that although the declared return type for method `Geom.createObject()` is `ObjectIn2D`, its definition reveals that it may return an object of any of the types: `Circle`, `Square`, or `EqTriangle`. Therefore, whether to propagate the declared return type of a

method, or the (set of) actual return type(s) presents a design option. Choosing to propagate the declared type is imprecise but convenient: since there is always only one declared return type, and it can be identified by merely inspecting the method signature. Whereas, in comparison, the actual return type(s) of a method definition may be many, and in order to find them, one must analyse the method definition. Although the second option leads to higher precision, it imposes an ordering on the analysis of the methods. Given that our invocation graph permits cycles, such ordering implies the need for a fixed-point computation. While this may be expensive, it is a choice that makes for a better analysis.

- Modelling the type fetched from field access: Notice that the object reference fetched as the return value from method `Geom.createObject()` is stored into class attribute `Goem.o`. Subsequently, methods `area`, and `perimeter` are invoked on the target object: `Geom.o`. Whereas the declared type of `Goem.o` is `ObjectIn2D`, the actual runtime type of the object referenced by it may be one of: `Circle`, `Square`, or `EqTriangle`. A design choice (similar to the one above) is presented by such usage: whether to propagate the declared type of the attribute, or the (compatible) type(s) of the object(s) that may have been stored at such location by some assignment operation(s). Propagating the declared type is convenient but imprecise: since there is only one declared type, and it is found by merely inspecting the attribute declaration. Whereas, in comparison, the options for the actual fetched type(s) from an attribute access may be many, and in order to find them, one must analyse *all* methods that assign into it. Therefore, although the second option leads to higher precision, it imposes an ordering on the analysis of the methods. Given that our invocation graph permits cycles, such ordering implies the need for a fixed-point computation. This may be expensive, but this choice that makes for a better analysis.

Our prototype implementation (Stalemate) chooses the ‘quick but imprecise’ option

in both cases with a view to catch the low hanging fruit. However, for more critical usage contexts the other option, 'precise but elaborate', may be more appropriate.

5.1 Purpose of Abstract Interpretation

Abstract interpretation of the method body associated with a node of the invocation graph is conducted with a view to establish the following. Of the four results listed hereunder, the first two are essential for conducting the analysis at both levels of rigour: 'quick but imprecise', as well as 'elaborate but precise'. The other two components of the computation are relevant only for the 'elaborate but precise' level of analysis.

- **Node-set for a method that contains `monitor` instructions**: If the method contains and use of `synchronized` statements, then corresponding to such usage, we want the invocation graph to have additional nodes. Each such additional node's `lockType` attribute has to identify the type of object locked before executing code within the scope of the corresponding `synchronized` statement. Also, containment edges (directed) connecting the node representing the 'containing scope' to the node representing the 'contained scope' are identified as a part of this computation.

  Typically, before its abstract interpretation, such a method would have been represented by a single node: that created during the initialisation of the invocation graph. Upon completion of its abstract interpretation, the invocation graph is updated: additional nodes are added to represent its `synchronized` statements, and outgoing directed edges are inserted from such nodes to represent invocations of other methods while holding the lock. Further, amongst the nodes of this set, containment edges are inserted as appropriate. The `lockType` attribute for all the non-root nodes of this set is non-null. Additionally, if the root node represents a `synchronized` method, then its `lockType` attribute would also be non-null.

- **The edges of the invocation graph**: We systematically collect and propagate the runtime types of objects that are both passed in as invocation arguments and also created by any `new` operations contained in the method body. As a result, accurate type information regarding all arguments passed to subsequent method invocations, as well as the type of the target objects is always available at call sites. Using this, the dynamic method dispatch can be reliably predicted. Such prediction is the basis for identifying the directed invocation edges that connect the nodes of the invocation graph.

- **Set of Return-types that a method invocation may fetch**: The invocation of a method may return only one object, however its actual type may be different for different invocations. We want to establish the set of all types that the invocation of a particular method behaviour may possibly return.

- **Set of types assigned into every attribute of every Class**: While propagation runtime types of objects, when the analysis encounters either `setfield`, or `setstatic` instruction that assigns a reference into an instance or class attribute (respectively), it takes a note of the type of reference assigned. When these findings are recorded during the analysis of all the method bodies, cumulatively, we fetch the mapping: $T_i.a_j \rightarrow \{T_k, T_l, ...\}, \forall types : T_i, \forall attributes : a_j$, where $T_k, T_l, ...$ are types subclassed from the declared type of $T_i.a_j$.

In effect, Abstract interpretation of a node of the invocation graph is conducted to discover the following relation, which defines the invocation graph:

$$< invSign, behSpec > \Longrightarrow < lockType, nextNodes, retTypes >$$

Where, *lockType* is the type of object locked before computing the corresponding code fragment, *nextNodes* is the set of all nodes of the invocation graph that are reachable from the given node through an invocation edge or through an containment edge, and

retTypes is the set of all runtime types of objects that an invocation of the given node may possibly return. Both *nextNodes*, and *retTypes* are overestimates: depending upon the values of the invocation arguments and the application state, some subset of *nextNodes* may not be invoked, and certainly an object of only one type from *retTypes* is actually returned. However, there would not exist any assignment of types to formal parameters that may cause invocation of any method outside of *nextNodes*, or cause the object returned to be of an actual type outside of *retTypes*.

In order to conduct abstract interpretations of a method body it is necessary to identify the precise scope of every `synchronized` statement in the method. Recall that the method bytecode contains precisely one `monitorenter` bytecode instruction for every source code use of `synchronized` statement. However there may be many `monitorexit` instructions that correspond to it, including the compiler inserted one that ensures the release of the lock if an exception is thrown during the execution of the statement. For our analysis it is important to identify the *last* `monitorexit` (in the bytecode) that corresponds to the closing brace of the corresponding `synchronized` statement (in the source). This needs to be ascertained for every `monitorenter` in the bytecode. We use the information available in the "Exception Table" for this purpose.

5.2 The scope of a SYNCHRONIZED statement

In this section, we illustrate using a real example how the contents of the exception table are used to fetch precisely the bounds of the scope for all the `synchronized` statements. The output of this computation is a list of pairs of offsets into the bytecode array that identify, correspondingly, the `monitorenter` and its last matching `monitorexit` instruction locations.

Before we start, let us re-visit the example bytecode in Section 1.4. Notice, firstly, that there is a single `monitorenter` but two `monitorexits`. The first `monitorexit`, at

bytecode offset 24, corresponds to the release of the monitor after normal execution of `this.doThat()`. The second one, at bytecode offset 27, corresponds to the release of the monitor in case an exception occurs during the attempt to compute `this.doThat()`. Also, notice that the "Exception table" has two entries that have '25' as the target, and 'any' as their 'Type'. The first such entry encapsulates the entire body of the `synchronized` statement between its 'from' and 'to' bytecode offsets. This is the entry we want to use to identify the *last matching* `monitorexit` corresponding to the `monitorenter` instruction. The second entry encapsulates bytecode introduced by the compiler to ensure the block structured semantics of the `monitor` instructions even in face of exceptional control flow. Our analysis discounts such sets of instructions; it takes cognisance of only such bytecode instruction sequences that correspond to program fragments.

More specifically: note that the 'from value' corresponding to this entry is one greater than the bytecode offset of the `monitorenter` that marks the beginning of the encapsulated code, and the 'to value' is one greater than the bytecode offset corresponding to the encapsulated code end. Below, we use a more interesting example of the "Exception table", to look at the details.

```
Exception table:
 from   to  target type
    6    54    65   any
   55    62    65   any
   65    69    65   any
   88    99   108   any
  102   105   108   any
  108   113   108   any
  116   123   126   Class java/lang/InterruptedException

  155   162   165   Class java/lang/InterruptedException

  148   173   176   any
  176   181   176   any
  192   202   205   any
  205   210   205   any
   79   227   230   any
  230   234   230   any
LineNumberTable:
```

From the above table, we firstly discount all entries that have the same value for the 'From' and the 'Target' column, while having 'any' in the 'Type' column, since they correspond to (exception handling) code inserted by the compiler. Next: we discount the entries that name some specific 'class' in their 'type' column, since they correspond to programmer specified exception handling code that is not relevant to identifying the *last matching* `monitorexit`. We are then left with only such rows, that all have, 'Any' in their last column. Dropping blank lines, header & trailer, and the last column, fetches us the following table:

```
  6    54    65
 55    62    65
 88    99   108
102   105   108
148   173   176
192   202   205
 79   227   230
```

When there is an 'if then else' statement as the encapsulated code, wherein the processing on one of the branches (either the then branch or the else branch) terminates with a return of control to the calling method, or based upon the length of the then part and the else part, there may be more than two `monitorexit` instructions corresponding to the `monitorenter`. All such entries in the "Exception table:" correspond to the 'normal processing' of the 'encapsulated code'.

Finally, starting from the top row, we start to merge two contiguous rows into one if their 'target' addresses are identical, and they meet a specific criterion: if the 'from' byte code address of the previous row identifies the first instruction following a `monitorenter`, and the 'to' address of the previous row, & the 'from' address of the next row identify contiguous byte code addresses in the method body, then the merged row has the 'from' address from the previous row, the 'to' address from the next row, the 'target' address that is same as that in both the rows. Thus merged rows correspond to different code paths within the same `synchronized` statement. In our example, af-

ter firstly merging rows where possible, and then dropping the last column, we are left with the 2 column table that maps (in its each row) the bytecode address following a `monitorenter` instruction on to the bytecode address following the one that identifies the `monitorexit` that closes the scope of the corresponding `synchronized` statement. We confirm that the offsets (into `code[]`) preceding the ones in the table below are the monitor enter & exit instructions, and store their offset values in `monitors` map.

```
  6    62
 88   105
148   173
192   202
 79   227
```

For the above example, therefore, the result of this computation is:

$$\texttt{monitors} \equiv [5 \Rightarrow 61, 87 \Rightarrow 104, 147 \Rightarrow 172, 191 \Rightarrow 201, 78 \Rightarrow 226].$$

Our prototype implementation (Stalemate) of the above described processing of the contents of the "Exception table:" is able to stand a system assertion that asserts for every method body, that either it doesn't contain any `monitorenter` instruction *iff* we have fetched an empty mapping, or that if we have fetched a non null mapping, then that it does indeed contain the `monitor enter/exit` instructions at the identified bytecode addresses as per the map entries.

5.3 Abstraction and Initialisation

Conceptually, bytecode execution uses the notion of an *operand stack*, and a *locals array* per method activation stack frame, the contents of which are used during instruction execution to manipulate *values* of variables. Abstract interpretation uses similar mechanisms, but manipulates *types* of variables instead of their *values*. It maintains a

type stack, and a *type array*, whose entries represent the type of the value that would correspondingly occupy the respective slot during program execution. The JVM Specification [7] requires that while the entries into the operand stack and the locals array be single word wide, access to them must be in strict accordance to their actual types. Our representation of the types of these entries and initialisation of abstract interpretation, is consistent with this requirement. The types are represented as follows:

- **Abstract types**: All single word basic types (`int`, `short`, `byte`, `char`, `boolean`, `float`) are modelled by one abstract type *c1*, and double word basic types (`long`, `double`) by another abstract type represented by 2 contiguous entries of *c2*.
- **Actual types**: Reference types are represented by their type signatures, and the `returnAddress` type used by the `jsr/ret` instructions is represented by the actual value that denotes the instruction address within the method's bytecode.
- **Array types**: We faithfully represent the dimension of an array type; its element type is represented as per points above: therefore `int[][]` gets represented as '[[c1', and `String[]` as '[Ljava/lang/String;'.

Initialisation: Depending upon whether a node represents method definition or its variant behaviour, its locals *type array* is initialised as discussed below. In either case, if the method is an instance method, then the type of the target object is used to populate the slot-0 of the locals *type array*. If however, the method is a class method, then the types representing arguments or parameters are populated starting from slot-0 of the locals *type array*.

- Nodes representing method definition: Interpreting such a node using the type information encoded in its method signature fetches the model of its behaviour when invoked exactly as defined. The invocation signature for such invocation would therefore be identical to the method signature of the node. In such case the

first few slots of the locals *type array* are initialised using the type of the defining class, and the types of all its formal parameters, appropriately abstracted as above, and in that order.

- Nodes representing (variant) method behaviour: In general, such a node represents behaviour of a method that is not invoked exactly as defined. Such a node is specified using the 2-tuple $< invSign, behSpec >$, where the bytecode corresponding to the method body is provided by *behSpec*, and the runtime types of the target object and the actual arguments are encoded as part of the *invSign*. For such nodes, therefore, the first few slots of the locals *type array* are initialised using the types of the target object and the parameters of the method as encoded in the *invSign*, appropriately abstracted as above, and in that order.

In all cases we start with an empty *type stack*, and the interpretation always starts with the first instruction in the bytecode sequence.

5.4 Interpretation

Generally, the semantics of abstract interpretation of the instructions is similar to the transition relation used by bytecode verification (indicated in [26]), with simplifications as afforded by our abstract representation of the primitive types. There are notable differences, given that our purpose is different. Interpretation starts with the first instruction in the bytecode sequence, using the initial values for the *type stack* & *array*. In general, abstract interpretation of an instruction uses the 'before' state of the type stack & type array to produce their 'after' state.

As it progresses, evolving the state of the stack & array one instruction at a time, two important & related subgoals have to be accomplished, as well: (1) to discover all the *sequentially executable instruction blocks (seib)* that compose the method body

including the various initial states of the *stack & array, combine* that may initialise them, (2) to accurately take note of the scopes of any `synchronized` statements used in the method. A *seib* is a maximal contiguous subsequence of instructions of the method bytecode that, once execution starts at its first instruction, it *may* sequentially continue until its last instruction, and not any further. Typically, the last instruction of the *sieb* can be like: *goto, jsr, ret, lookupswitch, tableswitch* or *return*. Usually, the first instruction of a *seib* is the instruction at offset 0, or the instruction that is target of some control transfer instruction.

The *seib*(s) are recorded in a set of 3-tuples: $seib \equiv < seibOffset, seibTs, seibTa >$. Each 3-tuple represents an initialisation for the interpretation of a sequentially executable instruction block: where, $Offset$ stores the offset of its first instruction in the bytecode array, and the Ts & Ta, combine are values used to initialise the 'before' state of its first instruction.

Interpretation of different categories of instructions are discussed below:

1. **Interpreting sequential instructions**: The 'after' state produced from interpreting non control transfer instructions serves as the 'before' state for the interpretation of the sequentially following instruction. Such include categories: arithmetic, load/store, array component load/store, stack operations & type conversion. For instance when faced with the need to interpret an 'iadd' instruction, we ascertain that the top two slots of `tS` indeed contain 'c1', each, and we pop both of them to symbolise their consumption. Subsequently we push 'c1' onto it, to symbolise the production of the result of its computation. For the 'dadd' instruction, we ascertain that the top 4 slots of `tS` contain 'c2', each. We 'consume' all of them, and then again push 'c2' onto it twice to symbolise the production of a double value from the computation. A more elaborate list of instruction types, actual examples, and our actions involved in their interpretation are listed in Table TC5.9, columns

1, 2 ,and 3, respectively.

2. `*newarray` **& array de-reference instructions**: Instructions to create or access arrays in the heap are subsumed in the category of sequential instructions discussed above. We discuss them separately here merely to point out that when interpreting them we faithfully represent the dimensional aspect. For instance if the bytecode uses `multianewarray` to create a 3-dimensional array of type `java.lang.String`, then its interpretation would push "[[[Ljava.lang.String;" onto the *type stack*. Correspondingly, interpretation of the de-reference operation for such an array fetches a type, one dimension less: "[[Ljava.lang.String;".

3. **Interpreting control transfer instructions**: Control transfer instructions include three categories: unconditional branch (ex. `goto`), conditional branch (ex. `ifnonnull`), and compound conditional branch (ex. `tableswitch`) instructions. The 'after' state produced by their interpretation can be used, as the 'before' state by one or more branch targets (including, possibly, the sequentially following instruction) as per the semantics of the specific instruction. Interpretation of such instructions contributes to the discovery of *seib*s that compose the method body, *along with* the various 'before' configurations that may initialise their computation.

   Consider a control transfer instruction like `iflt` that compares an integer value from the top of the operand stack with zero, and transfers control to the target bytecode index if the value is negative. There are therefore potentially two ways the control could flow after executing such an instruction: either it could continue with the next instruction in the sequence, or it could transfer to the target address. In either case, the state of the `tS`, and the `tA` would be just as the current instruction leaves it, for both code points. Interpretation of `iflt` firstly ascertains that there is 'c1' on the top of `tS` before consuming it. then a 3-tuple entry is inserted into *seib* with the target address as the value for `seibOffset`, and a clone of the current state of the `tS`, and `tA` objects as the other two fields of the entry. Having done

this interpretation continues with the sequentially next instruction.

4. **Interpreting method invocations**: Based upon the 'before' state, the invocation signature is composed. Then, in accordance with the semantics of the specific `invoke` instruction and in view of the compiler generated target method *methSign*, given the class & interface hierarchies, the *behSpec* is identified. A directed edge is inserted into the invocation graph from the current node to the node representing $< invSign, behSpec >$. In order to evolve the state of the *type stack* and *type array*, to reflect the interpretation of the invoke instruction, the parameters and the reference to the target object are popped off the *type stack* and a return type needs to be pushed on top of it. There are two options how to choose the return type: the easy option that makes for a quick but imprecise analysis is to propagate the declared return type of the *behSpec*, whereas in a more elaborate but precise analysis one must take cognisance of the possibility that the invocation may return any member from the set: *retTypes*. In case of the easy option, abstract interpretation continues from the subsequent instruction. The precise option is discussed below.

   Where the invocation offers the possibility of returning (amongst) multiple return types, then the remainder of the current *seib* ought to be interpreted for each of the return types that may result from the invocation. This scenario is handled as follows: the current *seib* containing the current invoke instruction is continued to be interpreted with one of the return type options, and the sequentially following instruction becomes the first instruction of the *next* set of *seib*s: initialising as many *type stack* configurations for the *next seib*, as many remnant return types may be returned from the invocation, with the corresponding return type pushed on top of it, instead of the declared return type.

   Similarly, when a target methods' invocation may throw exception that the invoking method has provided to *catch*, we treat such invocation as a potential termina-

tion of the current *seib*. Therefore, apart from the possibility that the sequentially following instruction may be executed next, we consider the possibility that the type stack be cleared, and re-populated with a single entry of the exception type thrown by the invoked method causing control to be transferred to the matching catch block.

5. **Interpreting monitor instructions**: Discovery of the type of the object locked by the `monitorenter` instruction is easily accomplished: it is conveniently present at the top of the type stack in its 'before' state. Interpreting `monitorenter` with every different value for the `lockType` causes a new node to be created in the invocation graph, and abstract interpretation continues in its context: all subsequent method invocations until encountering the last matching `monitorexit` causes directed edges to emanate from the corresponding node. This approach is used recursively for nested `synchronized` usage. Since a single `monitorenter` occurrence may have multiple matching `monitorexits`, we need to know which of them is the last matching one. Recollect that this is inferred automatically from analysing the corresponding Exception table.

   To enable handling of arbitrary nesting of `synchronized` statements, the interpretation maintains, locally, a stack (`nS`) of nodes. Upon entry, it initialises `nS` by pushing `this` (the reference to the current node) onto it. Subsequently, on encountering `monitorenter` when it creates the new node with the appropriate name, it pushes its reference onto `nS`. Later, on encountering the (*matching* `monitorexit` as confirmed by the `monitors` map, it 'closes' processing of the corresponding node of the call graph, by popping it off `nS`. When doing so, it takes care to add the containment edge to `nS.pop()` from `nS.peep()`. Any method invocations encountered in the meanwhile, (*i.e.* in between the `monitorenter` and its *matching* `monitorexit`), get recorded using directed edges originating from `oS.peek()`.

6. **Interpreting (`get\set`)(`static/field`) instructions**: These instructions that

read/write from/into the heap can be interpreted in two different ways: one quick but imprecise option, and another elaborate but precise option. The first option assumes that the analysis does not model the heap. In this approach interpreting an instruction that reads from heap is a lossy operation. During program execution (when the heap is present) a read operation would fetch the value saved by the most recent write to the same location. During abstract interpretation, however, the `put(static/field)` instruction reads off the top of type stack and discards the type (having no place/location to store it). Consequently, abstract interpretation of a `get(field/static)` instruction always pushes only the declared type of the accessed field/static member onto the *type stack*.

The other option makes for a precise analysis: in this approach the interpretation of every `put(static/field)` instruction reads off the top of type stack and stores the type so fetched into a set of types that is maintained per `static/field` attribute of every type defined in the JRE. Consequently, abstract interpretation of a `get(field/static)` instruction models the possibility that its execution may fetch an object of any of *that* set of types. Operationally this is affected as follows: the current *seib* containing the current `get(...)` instruction is terminated here, and the sequentially following instruction becomes the first instruction of the *next seib*. We then initialise as many *type stack* configurations for the *next seib*, as many different types may be assigned into the corresponding static/field attribute, with one, each, of the corresponding types pushed on top of it, instead of the declared type of the static/field attribute. Using this approach requires that whenever the interpretation of a `put` instruction does actually *grow* the set of types associated with the corresponding static/field attribute, all *behaviours* that involve a `get` on the same attribute would need to be re-interpret. This implies a fixed point computation that continues until sets corresponding to all static/field attributes stabilise.

7. **Interpreting exhaustively**: Abstract interpretation of a method invocation is complete only after all its *seib*s have been completely abstract interpreted starting correspondingly from each of their possible 'before' configurations.

In order to meet the above indicated design criterion, our implementation uses two methods to accomplish the task of abstract interpretation of an entire method body. The higher level method (`processMethodBody`) helps discover, and ensures interpretation of, all the various subsequences of the method body that *could* execute entirely if control were to enter it through its first instruction. (These subsequences are therefore not quite like the basic blocks in code compiled for RISC/CISC processors, where the semantics is that all its instructions *would* execute *iff* its first one did). The start points of these sub sequences are stored in the form of 3-tuples discussed above, and their union covers the entire method body (unless it contains dead or unreachable code). The other method (called `interpret`) implements the actual abstract interpretation of a single instruction at a time, given all the above described state, and advices its caller `processMethodBody` regarding the end point of sub sequences. The column 3 in Table TC5.9 contains actual java source fragments from the `interpret` method.

`processMethodBody` invokes `interpret` which actually updates the contents of `tS`, and `tA`, to reflect the abstract interpretation of the given instruction. `interpret` is invoked in a loop, every time handing it the sequentially next instruction to interpret. However this loop execution continues only so far as `interpret` continues to return `true`. For bytecode instructions like *[adfil]return* it returns `false`, to indicate that the execution trail terminates there, and the sequentially next instruction can not be analysed assuming the `tS`-state and the `tA`-state as the current instruction would leave it. Similarly `interpret` returns `false` for *goto(_w), jsr(_w), ret, lookupswitch, tableswitch* & *return*. For all other control transfer instructions, `interpret` returns `true`.

Consequently, the analysis of the method body happens in parts, starting from either

the beginning, or from the target of some control transfer instruction, and proceeds sequentially until `interpret` returns `false`. Every time this inner loop terminates, the next target of some control transfer instruction is read from the `seibOffset`, and the `tS` and `tA` are accordingly initialised. The loop so initialised continues the analysis of the corresponding sub sequence of the method body.

5.5 An Illustrative example

In this section, as an illustration of the discussion thus far, we examine how, for the code (file: `D.java`) depicted in Figure FC4.1 that defines Classes `A`, `B` and `D`, our analysis fetches the complete invocation graph depicted in Figure FC4.2. The command line: '`javac D.java`' is used to compile the code: the compiler generates three class files: `A.class`, `B.class`, and `D.class`.

The Class files generated by the compiler can be subject to analysis by issuing the command line: '`java JavaBlob -classpath . A.class B.class D.class`'. The execution of the tool is organised as two concurrent threads: one that reads the input classes and renders their contents into text, for consumption by another that initialises the invocation graph. The first thread fetches the text rendering of the input classes using the command: '`javap -c -private -verbose -classpath . A B D`'. Figure FC5.1 reproduces relevant portions of the so generated textual rendering: for brevity, we have removed the 'LineNumber Table', and the 'StackMap Table' portions associated with all method bodies. Similarly, the 'Constant Pool' for every class is also not included in the figure. In fact the other thread that initialises the invocation graph, for the most part, also discounts the missing portions of the input.

```
class A extends java.lang.Object
  SourceFile: "D.java"
{
public int i;

A();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public void setI(int);
  Code:
   Stack=2, Locals=2, Args_size=2
   0:   aload_0
   1:   iload_1
   2:   putfield        #2; //Field i:I
   5:   return

}
```

```
class B extends A
  SourceFile: "D.java"
{
public int ii;

B();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #1; //Method A."<init>":()V
   4:   return

}
```

```
class D extends java.lang.Object
  SourceFile: "D.java"
{
D();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public void gain(A);
  Code:
   Stack=2, Locals=6, Args_size=2
   0:   aload_1
   1:   dup
   2:   astore_2
   3:   monitorenter
   4:   aload_1
   5:   bipush  33
   7:   invokevirtual   #2; //Method A.setI:(I)V
   10:  aload_0
   11:  dup
   12:  astore_3
   13:  monitorenter
   14:  aload_1
   15:  bipush  77
   17:  invokevirtual   #2; //Method A.setI:(I)V
   20:  aload_3
   21:  monitorexit
   22:  goto    32
   25:  astore  4
   27:  aload_3
   28:  monitorexit
   29:  aload   4
   31:  athrow
   32:  aload_2
   33:  monitorexit
   34:  goto    44
   37:  astore  5
   39:  aload_2
   40:  monitorexit
   41:  aload   5
   43:  athrow
   44:  return
  Exception table:
   from   to  target type
    14    22    25   any
    25    29    25   any
     4    34    37   any
    37    41    37   any

public void plain(boolean);
  Code:
   Stack=2, Locals=3, Args_size=2
   0:   iload_1
   1:   ifeq    14
   4:   new     #3; //class A
   7:   dup
   8:   invokespecial   #4; //Method A."<init>":()V
   11:  goto    21
   14:  new     #5; //class B
   17:  dup
   18:  invokespecial   #6; //Method B."<init>":()V
   21:  astore_2
   22:  aload_0
   23:  aload_2
   24:  invokevirtual   #7; //Method gain:(LA;)V
   27:  return

}
```

Figure FC5.1: The text rendering for Classes A, B, and D, using ‘javap’.

5.5.1 Initialisation of the Invocation Graph

Notice in Figure FC5.1, that there are three portions: the top left contains the text rendering for A.class, the one below that is for B.class and then below that is the rendering for D.class that continues till the end of the right column. In general, the structure for each of these portions is similar: a header that contains the class name, its parent class name, and names of any interfaces its implements, and the name of the source file that defines it, followed by (blank line delimited) list of attribute definitions and method definitions enclosed within a single pair of braces. Notice further that there are two methods defined for Class A, one for Class B, and three for Class D. These numbers are, each, one more than the methods defined by the programmer in Figure FC4.1. The extra methods are the compiler generated null parameter constructors, one each for the three classes. Notice the structure for each of the method bodies: the first line contains the method header, the static string 'Code:' marks the beginning of the method bytecode, the first line of which contains the compiler computed values for `Stack` (the max depth of the operand stack), the `Locals` (the maximum number of local variables necessary to conduct the interpretation of the method bytecode), and `Args_Size` (the number of arguments to be passed for any invocation of the method).

The actual bytecode follows: it is a sequence of lines that begin with an integer value that identifies the index into the (binary representation of the) bytecode at which this particular instructions starts, demarcated by a ':', and then followed by the actual bytecode instruction and its arguments if any). This is optionally followed by a *significant comment* that starts with '//' and continues till the end of line. We call this comment *significant* since it contains the text rendering of the value from the constant pool that is referenced by the argument of the bytecode instruction. Its presence saves us from having to refer into the constant pool, altogether.

The method body is optionally followed by an 'Exception table:'. Where a Java

method contains a `try-catch` block or a `synchronized` statement, there the compiler generates an exception dispatch table for the method body. Notice in Figure FC5.1, such a table at the end of the method body for `D.gain:(LA;)V`.

The thread that is tasked with parsing the text rendering to initialise the invocation graph gets to see the input for Class A, followed by Class B and finally for Class D. From the Class header information it takes note of all the names of Classes and Interfaces, and all the relations amongst them: parent-child, implements, and extends. It makes note of which Classes are defined as being `abstract`. From the Class body it also takes not of all attributes that have user defined types (Class/Interface). For each defined method, that it encounters therein, it creates a node of the Invocations graph. Various attributes of the node are populated from reading the relevant information from the input. Methods whose behaviours are not defined, for instance `abstract` methods, or `native` methods or `interface` methods do not reflect as nodes of the invocation graph. Rather their names are included into the corresponding lists. All such 'look-aside' information regarding the type system underlying the set of input classes is collected and maintained. It is necessary and used when conducting the abstract interpretations of method bodies.

From the method header, its name, containing class, its formal parameter list are noted. Also whether it is `public`, or `synchronized`, and such other information is noted. Whether it may throw exceptions, and of what types is also noted. From its method body, the *stack-depth*, the length of the *locals array* are noted. The contents of the bytecode array itself is retained for every defined method as a two-tuple: $< Offset[], code[] >$. $Offset[]$ is the array that stores the integer values at the left of the ':' for all the instructions in the bytecode, and $Code[]$ stores, correspondingly, the entire string to the right of the ':', *i.e.* the actual bytecode instruction along with its arguments and including the optional significant comment, where present. Finally, when present, the contents of the 'Exception table' are processed as per Section 5.1 and the `monitors`

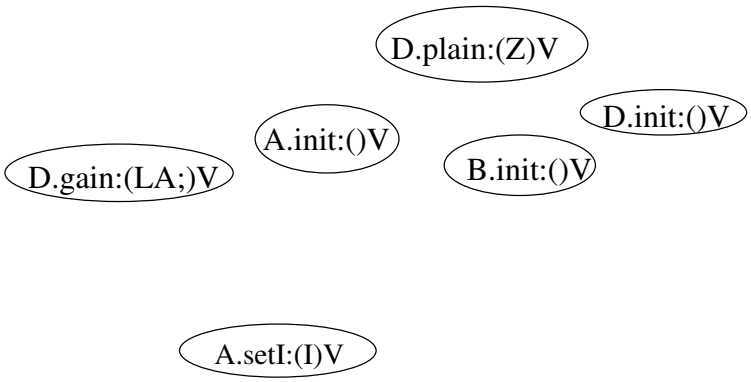

Figure FC5.2: Initial invocation graph for Classes A, B, and D from file: D.java

attribute of the node are populated.

Initialisation of the invocation graph is complete once all the input is processed. The initial invocation graph created for D.java is depicted in Figure FC5.2. Notice that the initial invocations graph consists of only nodes. No edges.

5.5.2 Abstract Interpretation of method 'D.plain:(Z)V'

As mentioned in section 5.3, the abstract interpretation of a method is overseen by a higher level method called `processMethodBody`, whereas the actual interpretation of every instruction is affected by the lower level method called `interpret` that does its job one instruction per invocation.

`processMethodBody` starts by initialising the state of the interpretation as follows: It inserts the 3-tuple $< 0, tS = [], tA = [LD;, c1,] >$ into the list of 3-tuples called `seib`. In that entry, 0 is the index of the first bytecode instruction, $tS = []$ is the empty stack, and the initial value for tA, an array of 3 entries: the $tA[0] = LD;$ corresponds to the type of the `this` pointer, and $tA[1] = c1$ is the abstract representation for the boolean argument. It does a `nS.push(this);` to initialise the nodeStack. Since the method body has no 'Exception table', its `monitors` attribute is empty. Its `offset[]`, and `code[]` arrays would already be appropriately initialised.

Table TC5.1: Interpreting 1^{st} seib of D.plain:(Z)V, seeded: $< 0, [], [LD;, c1,] >$

offset	code	intent – description	tS	tA
0	iload_1	tS.push(tA[1])	[] → [c1]	[LD;, c1,]
1	ifeq 14	if (tS.pop()==0) goto 14	[c1] → []	[LD;, c1,]
		action: seib.append($< 14, [], [LD;, c1,] >$)		
4	new #3; //class A	tS.push(new A())	[] → [LA;]	[LD;, c1,]
7	dup	tS.push(tS.peek())	[LA;] → [LA;, LA;]	[LD;, c1,]
8	invokespecial #4; //Method A.init:()V	invoke constructor	[LA;, LA;] → [LA;]	[LD;, c1,]
		action: nS.peek().nxtNodes.append(getNode("A.init:()V")		
11	goto 21	resume execution from 21	[LA;]	[LD;, c1,]
		action: seib.append($< 21, [LA;], [LD;, c1,] >$)		

Having been so initialised, the Table TC5.1 depicts how the interpretation of the first sequence of bytecode instructions evolves. Every row in the table corresponds to the interpretation of a single instruction in the sequence. The first and the second columns reproduce the offset and the bytecode instruction that is being interpret. The next column is descriptive: it mentions the 'intent' of the instruction. The last two columns reproduce the manner in which the 'before' state of the `tS`, and the `tA` variables, respectively, evolve to their 'after' states. Notice, that the 'after' state of `tS` and `tA` from the previous row are, respectively, their 'before' states in the next row. For all the rows except the last, the corresponding invocation of the `interpret` method returns `true`: signifying that the sequentially following instruction may be interpreted using the 'after' state as fetched from the current invocations of `interpret`. Where either `tS` (or `tA`) are not modified be the interpretation of the instruction, its 'before' and 'after' values are the same, and therefore mentioned only once. Finally, any additional significant action taken by the `interpret` invocation, for the current bytecode instruction is noted in the merged last three columns.

Interpretation of the first instruction in the Table TC5.1 causes the `c1` (abstract representation for `boolean`: the type of the argument) to be pushed on top of the (operand) type stack (`tS`). The next instruction's execution (depending on the value of the argument) *may* cause either transfer of control to the instruction at offset 14, or execution may continue sequentially. Interpretation of such instructions cause `seib` to be updated

to reflect such a possibility. In either case, the 'after' states of `tS` and `tA` at the end of the current instruction interpretation are used as their 'before' states, respectively. The instruction at offset 4 has a trailing 'significant comment' that identifies the type of object actually instantiated. Its interpretation causes this type to be pushed on top of `tS`. Subsequently `dup` merely pushes `tS.peek()` again on top of the stack.

The next instruction in the sequence, at offset 8 is a member of the `invoke` family. Its interpretation pops the types of the arguments to the invocation and that of the target object off `tS`, to first form the corresponding *invocation signature*. It then inserts a directed edge between the caller node and the called node of the invocation graph. The node corresponding to the called method behaviour may have to be created if not already present. As per the semantics associated with the specific member of the `invoke` family, (`invokespecial`, in this case), and in view of the type hierarchy constructed during the initialisation phase, the corresponding *behSpec* (where necessary) , as intended by the programmer is identified. Finally, the interpretation of this instruction pushes the return type of *behSpec* (if any) on top of `tS`, before returning `true`.

The last instruction in the sequence is a control transfer instruction: '`goto 21`'. Its interpretation updates `seib`, and returns `false`, thereby completing the interpretation of the current block of sequentially executable instructions. The next sequential block of bytecode instructions, from `seib`, are interpreted as tabulated in Table TC5.2, and described below.

The first instruction in the table is '`new`', with an intent to create an object of type `B`. Notice (from the code in Figure FC5.1) that the reference to this object is assigned to variable '`a`' whose declared type is '`A`': the super class of '`B`'. However, observe that the interpretation of this instruction pushes '`LB;`' on top of `tS`. It is the type that corresponds to the actual (runtime) type of the object rather than its (compile time) declared type. This is followed by a couple of instructions that arrange to invoke its

Table TC5.2: Interpreting 2^{nd} seib of D.plain:(Z)V, seeded: $< 14, [], [LD;, c1,] >$

offset	code	intent – description	tS	tA
14	new #5; //class B	tS.push(new B())	[] → [LB;]	[LD;, c1,]
17	dup	tS.push(tS.peek())	[LB;] → [LB;, LB;]	[LD;, c1,]
18	invokespecial #6; //Method B.init:()V	invoke constructor	[LB;, LB;] → [LB;]	[LD;, c1,]
		action: nS.peek().nxtNodes.append(getNode("B.init:()V")		
21	astore_2	tA[2]=tS.pop()	[LB;] → []	[LD;, c1,] → [LD;, c1, LB;]
22	aload_0	tS.push(tA[0])	[] → [LD;]	[LD;, c1, LB;]
23	aload_2	tS.push(tA[2])	[LD;] → [LD;, LB;]	[LD;, c1, LB;]
24	invokevirtual #7; //Method gain:(LA;)V	invoke virtual method	[LD;, LB;] → []	[LD;, c1, LB;]
		action: nS.peek().nxtNodes.append(new Node("D.gain:(LB;)V")		
27	return	exit	[]	[LD;, c1, LB;]

constructor (`B.init:()V`), similar to the initial part of Table TC5.1. Interpretation of the instructions `astore_2` at offset 21, and followed by instruction `aload_2` at offset 23 effectively *propagate* the runtime type information *collected* at offset 14, to the point of use, by the interpretation of `invokevirtual` at offset 24.

The interpretation of the `invokevirtual` instruction notices that although the compiler resolved the call against method '`D.gain:(LA;)V`' based upon the declared type of the reference variable `a` (and as witnessed from the significant comment), the actual (runtime) type of the argument of the invocation is `B` since the top of the type stack contains its signature. As required by the semantics of the `invokevirtual` instruction, given the type hierarchy, and in view of the above observation, the interpretation creates the invocation signature of the target of the invocation as '`D.gain(LB;)V`'. The type of the argument and the type of target object are both consumed from `tS`. Nothing needs to be pushed onto it since the invoked method return `void`. Interpretation of '`return`', the last instruction in the sequence closes this set of sequentially executable instruction block. Interpretation of the last sequentially executable instruction block for this method is tabulated in Table TC5.3, and described below.

Interpretation of the sequence of instructions between offsets 21 and 27 has already been once done as per the (trailing part of) Table TC5.2. However, notice that the 'before' value of the `tS` component of the state of the interpretation before interpreting the

Table TC5.3: Interpreting 3^{rd} seib of D.plain:(Z)V, seeded: $< 21, [LA;], [LD;, c1,] >$

offset	code	intent – description	tS	tA
21	astore_2	tA[2]=tS.pop()	[LA;] → []	[LD;, c1,] → [LD;, c1, LA;]
22	aload_0	tS.push(tA[0])	[] → [LD;]	[LD;, c1, LA;]
23	aload_2	tS.push(tA[2])	[LD;] → [LD;, LA;]	[LD;, c1, LA;]
24	invokevirtual #7;	invoke virtual method	[LD;, LA;] → []	[LD;, c1, LA;]
	//Method gain:(LA;)V	action: nS.peek().nxtNodes.append(getNode("D.gain:(LA;)V")		
27	return	exit	[]	[LD;, c1, LA;]

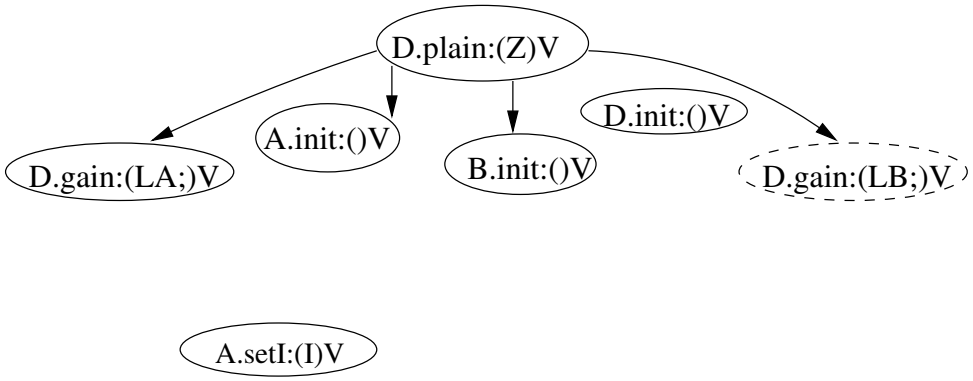

Figure FC5.3: Invocation graph after interpretation of D.plain:(Z)V

instruction at offset 21, in the two cases is different: in Table TC5.2 it is [LB;], whereas in Table TC5.3 it is [LA;]. Notice how this translates into the creation of *two* directed edges from the *current* node, both of which correspond to a single method invocation in the Java Source. But since the invoked behaviour can *potentially* be different depending upon the value of the boolean argument, this way of discovering reach-ability amongst behaviours of code fragments defined in the input creates an invocation graph that is sufficiently accurate for our purpose: that of identifying lock order violations.

After completing the abstract interpretation of method '`D.plain:(Z)V`', additional edges and nodes are created in the invocation graph that evolves it to the state depicted in Figure FC5.3.

Table TC5.4: Interpreting A.init:()V, seeded: $< 0, [], [LA;] >$

offset	code	intent – description	tS	tA
0	aload_0	tS.push(tA[0])	[] → [LA;]	[LA;]
1	invokespecial #1; //Method java/lang/Object.init:()V	invoke Object constructor	[LA;] → []	[LA;]
		action: nS.peek().nxtNodes.append(new Node("java.lang.Object.init:()V")		
4	return	exit	[]	[LA;]

Table TC5.5: Interpreting B.init:()V, seeded: $< 0, [], [LB;] >$

offset	code	intent – description	tS	tA
0	aload_0	tS.push(tA[0])	[] → [LB;]	[LB;]
1	invokespecial #1; //Method A.init:()V	invoke parent constructor	[LB;] → []	[LB;]
		action: nS.peek().nxtNodes.append(getNode("A.init:()V")		
4	return	exit	[]	[LB;]

5.5.3 Abstract Interpretation of the default constructors

The default constructors are all small, straight line sequence of bytecode instructions. The Table TC5.4 tabulates the interpretation of '`A.init:()V`', and Table TC5.5 tabulates the interpretation of '`B.init:()V`'. Notice from Table TC5.4 that the interpretation of the `invoke` instruction therein causes a new node to be created corresponding to the called method: '`java.lang.Object.init:()V`'.

The interpretations of '`D.init:()V`', proceeds very similar to that of '`A.init:()V`'. After completion of the abstract interpretations of these three compiler generated default constructors, additional nodes and edges are created in the invocation graph that evolves to the state depicted in Figure FC5.4.

5.5.4 Abstract interpretation of method 'D.gain:(LA;)V'

The Table TC5.6 tabulates the interpretation of the first seib of '`D.gain:(LA;)V`' that is seeded with the 3-tuple: $< 0, [], [LD;, LA;,] >$. The second sequential block of bytecode instructions are interpreted as tabulated in Table TC5.7. Finally, the last sequential block of bytecode instructions (actually comprising of only one instruction: `return`') are interpreted as tabulated (for completeness) in Table TC5.8.

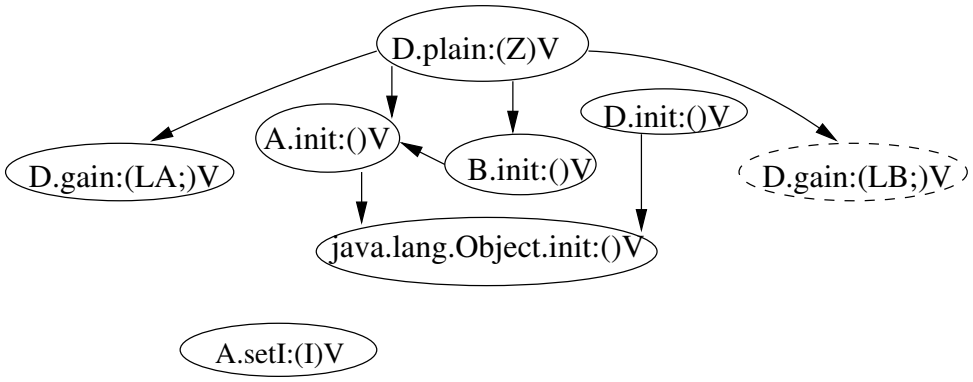

Figure FC5.4: Invocation graph after interpretation of default constructors

Table TC5.6: Interpreting 1^{st} seib of D.gain:(LA;)V, seeded: $< 0, [], [LD;, LA;,] >$

offset	code	intent – description	tS	tA
0	aload_1	tS.push(tA[1])	[] → [LA;]	[LD;, LA;,]
1	dup	tS.push(tS.peek())	[LA;] → [LA;, LA;]	[LD;, LA;,]
2	astore_2	tA[2]=tS.pop()	[LA;, LA;] → [LA;]	[LD;, LA;,] → [LD;, LA;, LA;,]
3	monitorenter	lock: tS.pop()	[LA;] → []	[LD;, LA;, LA;,]
		action: nS.push(new Node("D.gain:(LA;)V#LA;#0")		
4	aload_1	tS.push(tA[1])	[] → [LA;]	[LD;, LA;, LA;,]
5	bipush 33	tS.push(33)	[LA;] → [LA;, c1]	[LD;, LA;, LB;,]
7	invokevirtual #2; //Method A.setI:(I)V	invoke method	[LA;, c1] → []	[LD;, LA;, LB;,]
		action: nS.peek().nxtNodes.append(getNode("A.setI:(I)V")		
10	aload_0	tS.push(tA[0])	[] → [LD;]	[LD;, LA;, LA;,]
11	dup	tS.push(tS.peek())	[LD;] → [LD;, LD;]	[LD;, LA;, LA;,]
12	astore_3	tA[3]=tS.pop()	[LD;, LD;] → [LD;]	[LD;, LA;, LA;,] → [LD;, LA;, LB;, LD;,]
13	monitorenter	lock: tS.pop()	[LD;] → []	[LD;, LA;, LA;, LD;,]
		action: nS.push(new Node("D.gain:(LA;)V#LA;#0#LD;#1")		
14	aload_1	tS.push(tA[1])	[] → [LA;]	[LD;, LA;, LA;, LD;,]
15	bipush 77	tS.push(77)	[LA;] → [LA;, c1]	[LD;, LA;, LA;, LD;,]
17	invokevirtual #2; //Method A.setI:(I)V	invoke method	[LA;, c1] → []	[LD;, LA;, LA;, LD;,]
		action: nS.peek().nxtNodes.append(getNode("A.setI:(I)V")		
20	aload_3	tS.push(tA[3])	[] → [LD;]	[LD;, LA;, LA;, LD;,]
21	monitorexit	Unlock: tS.pop()	[LD;] → []	[LD;, LA;, LA;, LD;,]
		action: nS.pop()		
22	goto 32	continue from 32	[]	[LD;, LA;, LA;, LD;,]
		action: seib.append($< 32, [], [LD;, LA;, LA;, LD;,] >$)		

Table TC5.7: Interpreting 2^{nd} seib of D.gain:(LA;)V, seeded: $< 32, [], [LD;, LA;, LA;, LD;,] >$

offset	code	intent – description	tS	tA
32	aload_2	tS.push(tA[2])	[] → [LA;]	[LD;, LA;, LA;, LD;,]
23	monitorexit	relinquish lock	[LA;] → []	[LD;, LA;, LA;, LD;,]
		action: nS.pop()		
34	goto 44	resume execution from 44	[]	[LD;, LA;, LA;, LD;,]
		action: seib.append($< 44, [], [LD;, LA;, LA;, LD;,] >$)		

Table TC5.8: Interpreting 3^{rd} seib of D.gain:(LA;)V, seeded: $< 44, [], [] >$

offset	code	intent – description	tS	tA
44	return	exit	[]	[LD;, c1, LA;]

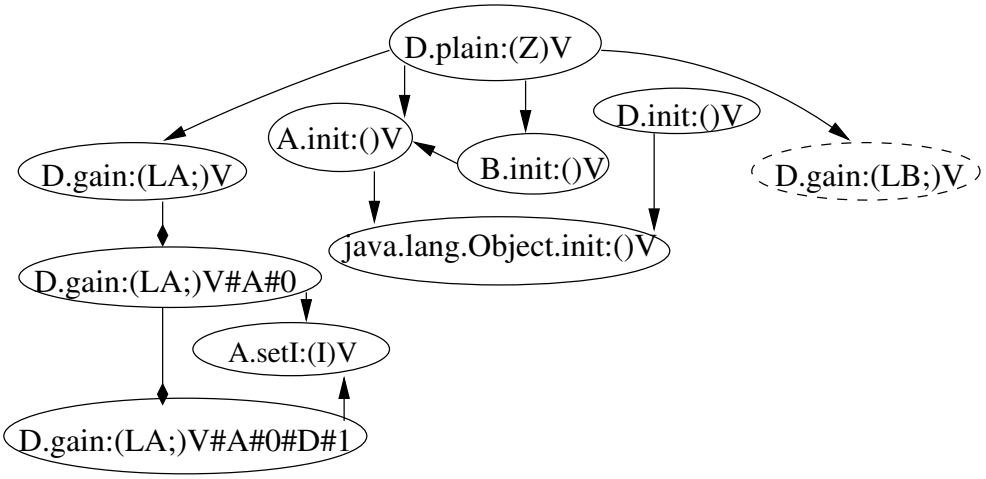

Figure FC5.5: Invocation graph after interpretation of D.gain:(LA;)V

After completing the abstract interpretations of method '`D.gain:(LA;)V`' as discussed above, additional edges and nodes are created in the invocation graph that evolves to the state depicted in Figure FC5.5. Finally, after abstract interpretation for the node depicted as '`D.gain:(lb;)V`' that is short for the full name: `D.gain:(lb;)V-varientOfAndcompiledAs-D.gain:(LA;)V`' the invocation graph evolved to its fully evolved form as depicted in Figure FC4.2. This last interpretation proceeds quite like the discussion in this subsection, except that it is initially seeded by the 3-tuple: $< 0, [], [LD;, LB;,] >$ that leads it to create new nodes of the subgraph rooted thereunder.

5.6 More examples of the transfer function

The previous section contains a good number of examples of how specific bytecode instructions are interpreted by our approach. However, they are all bytecode sequences that do not cover the entire family of bytecode instructions. The Table TC5.9 below

lists some more of the instructions that are not already covered above.

5.7 Comparison with Bytecode Verification

The JVM conducts 'bytecode verification' [27] when loading a class during program execution, which also involves type level abstract interpretation for all the methods of the class. Our abstract interpretation of the method body uses a similar transition relation in the same spirit as does the JVM bytecode verification step. For instance, in doing so, our data structures (type stack & type array) are quite similar. However, bytecode verification by the JVM is primarily to *incrementally* ascertain type consistency & safety, and the well-formedness of classes loaded into an executing application. Whereas our intent is, primarily, to accurately estimate the runtime types of objects that compose the calling context for method invocations, and the types of objects locked by the `monitorenter` instructions.

Bytecode verification for a method body identifies control join points, where its attempt is to identify a *single* type that needs to be propagated from there forward. Their approach discards runtime type information or dilutes it to ensure, merely, that the code downstream of a control join point is legal. Contrary to that, our analysis collects and propagates runtime type information, since our abstract interpretation of the method body re-interprets its every *seib* with every different combination of initial states for the type stack & array as can potentially manifest at run time.

Summary: This chapter discussed abstract interpretation of the method body. It starts by presenting an example to demonstrate how it may be conducted either in an 'imprecise but convenient' way, or in a 'precise but elaborate' way. The outcomes computed as results of abstract interpretation and their significance in connecting nodes of the invocation graph are listed.

Table TC5.9: Abstract interpretation for some Java bytecode instruction categories

Instruction Description, and specification	Actual example from the JRE	Semantics
not an operation	nop	–
Unary Arithmetic: [ilfd]neg	ineg	assert (tS.peek()==c1)
	dneg	assert (tS.pop()==c2); assert (tS.peek()==c2); tS.push(c2);
Load a local onto stack: [adfil]load_[0123], [adfil]load	aload_3	tS.push(tA[3]);
	dload 7	assert (tA[7]==c2); tS.push(c2); assert (tA[8]==c2); tS.push(c2);
Store top of stack into a local: [adfil]store, [adfil]store_[0123]	istore_2	assert ((tA[2]=tS.pop())==c1);
	dstore 6	assert ((tA[6]=tS.pop())==c2); assert ((tA[7]=tS.pop())==c2);
Load constant onto stack: [bs]ipush, iconst_m1, fconst_[012], aconst_null, [dl]const_[01], iconst_[012345]	sipush 931	tS.push(c1);
	dconst_0	tS.push(c2); tS.push(c2);
	dconst_0	tS.push(c2); tS.push(c2);
	aconst_null	tS.push("Ljava/lang/Object;");
Numeric Conversions: [ilfd]2[bsilfd]	i2l	assert (tS.pop()==c1); tS.push(c2); tS.push(c2);
	d2f	assert (tS.pop()==c2); assert (tS.pop()==c2); tS.push(c1)
Binary Arithmetic: [ilfd] ((add)|(sub)|(mul)|(div)|(rem)) Bitwise: [il]((or)|(xor)|(and)), Shift: [il]sh[lr], [il]ushr Comparison: [fd]cmp[gl], lcmp	fadd	assert (tS.pop()==c1); assert (tS.peek()==c1);
	ixor	assert (tS.pop()==c1); assert (tS.peek()==c1);
	lshr	assert (tS.pop()==c1); assert (tS.pop()==c2); assert (tS.peek()==c2); tS.push(c2);
	fcmpg	assert (tS.pop()==c1); assert (tS.peek()==c1);

Load array component onto stack: [bcsilfda]aload	caload	assert (tS.pop()==c1);
		String aN = tS.pop(); tS.push('c1');
		assert (aN=='[...') || aN=='Ljava/lang/Object;');

Store top of stack into array component: [bcsilfda]astore	iastore	assert (tS.pop()==c1); assert (tS.pop()==c1);
		String aN = tS.pop();
		assert (aN=='[...') || aN=='Ljava/lang/Object;');

Get length of array: arraylength	arraylength	String aN = tS.pop(); tS.push (c1); assert (aN=='[...') || aN=='Ljava/lang/Object;');

Stack operations: ((pop)|(dup))2?, dup_x[12], dup2_x[12], swap	dup_x1	String ts=tS.pop(), tsn=tS.pop(); tS.push (ts);
		assert (ts!=c2 && tsn!=c2);
		tS.push(tsn); tS.push(ts);

Conditional branch: if(non)?null, if((ne)|(eq)|([lg][te])), if_icmp((ne)|(eq)|([lg][te])), if_acmp((ne)|(eq))	ifle 1420	assert (tS.pop()==c1); seib.add (1420, tS.clone(), tA.clone());

	if_acmpeq 118	String ts=tS.pop(), tsn=tS.pop();
		assert (ts!=c1 && ts!=c2 &&
		tsn!=c1 && tsn!=c2);
		seib.add (118, tS.clone(), tA.clone());

Compound cond. branch: tableswitch, lookupswitch	lookupswitch{ //1 28: 24; default: 29 }	seib.add (24, tS.clone(), tA.clone()); seib.add (29, tS.clone(), tA.clone());
Unconditional branch: ret, goto(_w)?, jsr(_w)?	jsr 303	tS.push (nextInstrOffset); seib.add (303, tS.clone(), tA.clone());

Other instructions: athrow, wide, iinc, ldc_w, ldc2(_w)?	ldc_w #220; //class..
	..java/net/Socket
		String cObj = 'java/net/Socket';
		tS.push ('L' + cObj +'.class;');

Table TC5.10: Semantics of abstract interpretation for Java bytecode instructions

Instruction Description, and specification	Actual example from the JRE	Semantics
Create an instance: new	new #200; //class.. ..java/lang/String	String tN = "java/lang/String"; //read type name tS.push('L' + tN + ';'); // push its signature
Create an array: newarray, anewarray, multianewarray	newarray char	assert (tS.pop()==c1); //consume the array size tS.push('[' + c1); // 'char' encodes to c1
Method Invocation: invoke((special)|(static))| invoke((interface)|(virtual))	invokestatic #101; //Method.. ..nextIndex:(II)I	assert (tS.pop()==c1); assert (tS.pop()==c1); //use 2 args oS.peek().noteCallTo (classNm + '.' + 'nextIndex:(II)I'); tS.push (c1); //push return type
Access Class/Instance Field: ((get)|(put))((field)|(static))	getfield #356; //Field value:[C	String tN = "[c1"; //*rhs* of the ':' in the tS.push(tN); // significant comment
Check Type properties: instanceof, checkcast	instanceof #51; //class java/util/Map	String ts=tS.pop(); tS.push(c1); assert (ts!=c1 && ts!=c2);
Various returns: ret, return [ilfda]return	ret 4	seib.add (tA[4], tS.clone(), tA.clone());
	dreturn	assert (tS.pop()==c2); assert (tS.pop()==c2);

How to demarcate the scope of `synchronized` blocks is discussed in detail with an example. The abstraction, initialisation, and the interpretations is discussed in detail. An example is included to illustrate the various steps involved in abstract interpretation of real code. More examples of the transfer function are included for completeness. Finally the notable distinctions between abstract interpretation of the method body (described in this chapter) and the bytecode verification (as conducted by the JVM during class loading) are highlighted.

The next chapter describes the type lock order graph, and the significance of its nodes, edges, and cycles.

CHAPTER 6

TLOG AND LOCK ORDER VIOLATIONS

Contents: This chapter describes the type lock order graph (tlog): its nodes, edges, and their significance. Cycles in the tlog correspond to lock order violations in the input. A concise reporting of these helps identify sites in the source that programmers could review. This chapter is a rigorous treatment of the graph created in Step 3 of Section 3.3.

The type lock order graph (tlog) represents, collectively, the lock-orders discovered along paths (in the invocation graph) reachable from the various public entry points into the library. The structure of this graph, and the semantics associated with its nodes is briefly described in section 3.2. In our early report [23] of results from an initial prototype, we called it the Lock Acquisition Graph (LAG). However, we believe that **tlog** is a more appropriate name. This chapter describes it completely: its design & construction, the significance of cycles in it and the concise reporting of collections of related cycles in the form of 'lock order violations'.

6.1 Nodes of the *tlog*

In general, nodes of the *tlog* represent locks acquired by fragments of code (or their behaviour) represented as nodes in the invocation graph. Whenever the execution of

a code fragment requires an object to be locked, the `lockType` attribute of the corresponding node of the invocation graph records the type of the locked object. Further, if the invocation graph contains a node whose `lockType` attribute has value `X`, then to represent the possibility that an object of type `X` may be locked, a node named '`X`' is created in the *tlog*. Contrarily, if objects of a type are not locked by any node of the invocation graph, then such a type is not represented in the *tlog*.

More specifically, the *tlog* comprises of two kinds of nodes: the ones that, typically, represent the types of the object locked by instance methods, and another that represent locks taken, typically, by `static synchronized` methods. The above description is of the first kind of locks, and the second kind of nodes are discussed below.

Consider a Class `Y` that defines a `static synchronized` method member, say, `yy`. During the execution of method `yy`, the `java.lang.Class` object corresponding to the type `Y` is locked. To represent this lock acquisition, if the general guideline described above were to be followed, then the `lockType` attribute of the node '`Y.yy`' of the invocation graph would have value `java.lang.Class`. In fact, using this approach, the `lockType` attribute of all nodes of the invocation graph that correspond to the definition or behaviour of `static synchronized` methods would have the same value: `java.lang.Class`. This would lead to a significant loss of information resulting in either far fewer lock-order cycles being detected by our analysis and/or an undesirable increase in the proportion of false positive reports. To maximise the usefulness of our analysis, we choose to explicitly represent every *object* of type `java.lang.Class` that may be locked by any node of the invocation graph.

The possibility that the `java.lang.Class` object corresponding to class `Y` would be locked during the execution of `Y.yy`, for instance, is therefore represented by populating the `lockType` attribute of the corresponding node with '`Y.class`'. Contrarily, if the invocation graph does not contain any node that locks the `java.lang.Class` object

corresponding to some type (say, `Z`), then the node '`Z.class`' is not included in the *tlog*.

Briefly, therefore:

$$L \in tlog \implies (\exists beh \in InvGraph \bigwedge beh.lockType = L)$$

6.1.1 Abstract Interpretation of the expression: 'tN.class()'

Consider the usage: `synchronized (org.omg.CORBA.TypeCode.class) { ... }`. While translating such usage, the compiler first arranges to store a reference to the `java.lang.Class` object corresponding to the named class: '`org.omg.CORBA.TypeCode`' into the constants table of the containing class. The bytecode generated by the compiler corresponding to the expression in the parenthesis contains a sequence of the `ldc_w` instruction (to fetch the referenced `Class` object onto the top of stack), followed by a `monitorenter` instruction. When interpreting the `monitorenter` instruction, if our analysis were to discover the type on top of the type stack correctly as being *merely* `java.lang.Class`, then we would loose the information about exactly which type the object corresponds to. Instead, we detect the code-idiom of an `ldc_w` followed by a `monitorenter`, and accurately record the *type* whose '.class' is being locked. In this case, therefore we record it as '`org.omg.CORBA.TypeCode.Class`'

6.2 Edges of the *tlog*

In general, whenever the invocation graph contains a path from node '`N.m1:()V`' to node '`M.mn:()V`', both of which represent `synchronized` instance methods, then their `lockType` attributes would be assigned values: `N` and `M` respectively. The path '`N.m1:()V-...-M.mn:()V`' implies a type lock ordering amongst the types `N` and `M`.

Corresponding to such a path in the invocation graph, a directed edge is inserted into *tlog*, from node `N` to node `M`, with the '`N.m1:()V-...-M.mn:()V`' as the edge annotation.

More specifically, however, not *all* lock orders implied by *all* paths in the invocation graph get recorded into the tlog. For instance, in order to keep the number of reported lock order violations low we impose the following constraint: Only paths from the invocation graph that start and end on nodes with a non-null `lockType`, but with en-route nodes with a null-valued `lockType` attribute are permitted to correspond to edges of the *tlog*. Using this approach, our initial runs of the analysis on the JRE fetched a large number of lock order violations that included the `java.lang.Object` as one of the types. To improve the information content in the output of the analysis, we made the following provision in our definition of a contributing path: we permitted en-route methods that acquire locks of type `java.lang.Object`.

Consider a path 'C' in the invocation graph, that (starting from some `public` concrete entry point in the library) acquires three locks, on objects of types, T1, T2, and T3, in that order. Strictly speaking, C then implies a lock order amongst every two of the three types. Our prototype implementation, Stalemate, however chooses to record only the edge <T1, T2> into the tlog. It doesn't record the other edges <T2, T3>, or <T1, T3>. Alternatively, in the exception case, if T2 happens to be "java.lang.Object", then it records only the edge <T1, T3>, and none of the others. In spite of such "Under-recording" of actual lock-orderings implied by the invocation graph, our analysis ends up listing a very large number of lock order violations. Such under-recording is not intrinsic to the analysis; and ideally, it must be undone. However, since (in most cases) only the initial prefixes of the paths from the invocation graph are so used to infer lock-orders and report violations, the realisability of the reported call stacks is likely to be better. The justification for the exception, when (T2 == '`java.lang.Object`'), is to avoid recording edges into the tlog which may participate in cycles, but aren't much

help to the library maintenance engineer to improve his/her understanding of the library significantly. The intuition is that as thread stacks (fetched from static analysis, such as ours) get longer, their realisability keeps diminishing.

In spite of the (above) constrained definition of the edge of the *tlog*, we do fetch quite a large number of lock order violations.

6.3 Cycles in *tlog* and Lock Order Violations

After completing the construction of the invocation graph, the nodes for the *tlog* are created. Thereafter, all public concrete behaviours in the invocation graph are selected and the corresponding set of paths in the subgraph rooted at such behaviours are explored to identify which of them contribute to edges in the *tlog*. After populating all the edges in the *tlog*, the analysis proceeds to identify cycles and uniquely report them.

If, between nodes T_1 and T_2 of the *tlog*, there are directed edges $< T_1, T_2 > \equiv E_{cs1}$ in one direction and $< T_2, T_1 > \equiv E_{cs2}$ in the other direction, then we observe the following:

- **Violation of lock ordering**: Since the library offering is a collection of types together with their methods, and is intended for use by concurrent applications, it ought not be able to demonstrate both call stacks E_{cs1} and E_{cs2} along which the types T_1 and T_2 are locked in cyclical order.
- **Deadlock Possibility**: If an application thread t_1 invokes entry into call stack E_{cs1} and another thread t_2 concurrently invokes entry into call stack E_{cs2}, and if they share the two objects of types T_1, and T_2, then it is possible that such execution of threads t_1 and t_2 may deadlock.
- **Violations protected by a gate-lock** If, however, all public entries into the Library/API that can allow access to E_{cs1} and E_{cs2} are such that they *all* acquire

some lock of type (say) `T`, then it may be that the shared lock type `T`, could be a gate-lock that serialises access to the (lock order) violating stacks. Our analysis detects this scenario, and avoids reporting such cycles whose *concurrent* realisability is (possibly) protected by some gate-lock.

Stalemate reports all cycles in the *tlog* involving upto n nodes, where n is configurable (default value: 4). For a k-subset of types T_1, T_2, ... T_k, all k-cycles involving objects of these types are reported together. It contains the following information in XML format: $< T_1, ...T_i, ...T_k > \equiv Ts_1, \ ...Ts_i, ...Ts_k$, where $T_i, \forall 1..i..k$ are names of nodes of the *tlog*. In general, Ts_i lists the edge annotations of all directed edges $< T_i, T_{i+1} >$. Finally, Ts_k lists the edge annotations of all directed edges $< T_k, T_1 >$

Summary: This chapter described the type lock order graph, its nodes, edges, and their significance. Cycles in the tlog, and lock order violations are also described.

The next chapter discusses the results of the analysis of JRE, 1.6.

CHAPTER 7

RESULTS

Our prototype implementation, 'Stalemate' [28], is able to analyse the entire JRE, version 1.6.0_35, in under 10 minutes on a stock desktop: Dual CPU m/c with 4GB RAM, running GNU/Linux. The CPUs are Intel Pentium dual-core clocked @ 2.7GHz.

Table TC7.1 reports the size of the input: the number of Classes, and interfaces it defines. The count (Class) 'Inheritance relations' is fetched from computing the transitive closure of the `extends` relations between Class definitions. Similarly, the transitive closure of the `extends` relations between Interfaces, and the `implements` relations between Classes and Interfaces is computed to fetch the count of 'Implements relations' of the type: Class `implements` Interface. The breakup of the method definitions is in the right hand column. 652 of these are `static synchronized` methods, and 2681 are (instance) `synchronized`. The JRE contains 2963 instances of `synchronized` statements inside method definitions. The declarations of the `native` methods, and Interface

Table TC7.1: Size of the input

JRE Version: 1.6.0_35		Method	counts
Classes:	16,500	Synch. instance:	2,681
Interfaces:	2,300	Synch. static:	652
Inheritance Relations:	35,000	`public` concrete:	61,500
Implements relations:	22,000	Total defined:	1,60,000
		Undefined:	431
		Native:	1927
Synch. Statements:	2,963	I/f or Abstract:	16,734

Table TC7.2: Details of the Analysis & the Output

Invocation Graph Size	Total Nodes:	2,20,000	Nodes with Multiple `areturn`-types:	1383	Edges:	5,84,859
tlog size, & size of output	Nodes representing objects of `java.lang.Class`:	513	Nodes representing other object types:	1170	Total Nodes in tlog	1683
	Edges in tlog:	1,52,856	Cycles in tlog:	1,85,130		
Lock order Violations	Involving 2 types:	53	Involving 4 types:	529	Total Lock Order Violations:	714
	Involving 3 Types:	132				

or Abstract methods were taken note of: their behaviours are not defined in the input. Apart from these, the JRE also contains invocations to 431 methods, whose definitions *we* could not find.

The numbers in Table TC7.2 reflect the size of the analysis. In its completely evolved state, where all nodes representing reachable behaviours from `public` concrete entries into the JRE are explored, the invocation graph contains 2,20,000 nodes, with 5,84859 directed edges connecting them. Of the nodes, since 1,60,000 correspond to defined methods, most of the rest 60,000 nodes correspond to behaviours fetched from invocations, whose *invSign* differs from the compiler resolved *methSign*. Notice that the JRE exercises 1383 behaviours (based upon whose definitions, we find) that they may return objects of runtime types sub-classed from their declared return type. After the invocation graph is initialised by reading the JRE, we choose these 1383 methods (whose subsequent interpretation may invalidate the reachability computed for their callers, since they may return type(s) different from their declared return types), and conduct their abstract interpretation first. Thus minimising the need to re-compute the reachability of their callers. Therefore, effectively, bulk of the 2,20,000 nodes are abstract interpret only once, each.

At the end of the analysis, 1683 nodes compose the *tlog*: Of these, 513 nodes represent the different `java.lang.Class` objects that correspond to the Classes that contain some `static synchronized` method or statement. Each of the other 1170 nodes represents a type whose some object would be locked by some behaviour in the invocation graph. More than 1,50,000 annotated directed edges connect these tlog nodes.

Finally the analysis prints a total of 714 sets of lock order violations that form 1,85,000 cycles in the *tlog*: of these, 53 are 2-cycle, 132 are 3-cycle, and the rest 529 are 4-cycle violations. 614 violations involve types bundled in some earlier release that, surprisingly, invoke methods that belong to types released in a later release of Java.

7.1 From lock order violations to Deadlocks

Using the information in some of the lock order violations we investigated the relevant source files of the JRE and succeeded in developing 15 programs that could realise one of the cycles in the corresponding lock order violation leading to a deadlock during execution. Table TC7.3 lists the sets of types involved in the deadlock and the bugid[1] we have filed. All lock order violations reported for the Security, CORBA, and the Sound APIs were realised. Two bugs (In Table TC7.3: one[2] in the Security API(2nd row), and another in the J2SE API(last row; also refer Section 7.5)are fixed in recent releases of the JRE version 1.7.

We briefly discuss our learning from trying to use the reports from the analysis to develop small applications that can deadlock the executing JRE as predicted. The intent being to send such bug reports to bugs.sun.com who can then fix them in future JDK releases along various trains and thereby improve the platform reliability.

Every report produced from our analysis identifies specifically two *sets* of locks; any one lock from *each set* can be acquired using library code, in a possibly (mutually) deadlocking manner. Favourable conditions *necessary* for a program execution to realise such a deadlock include:

[1]These ids were fetched when we filed the bugs with Sun/Oracle. It is our observation that the actual web-link used to access the details of bug, using the id, has changed a few time so far. Moreover, the server may not be available all the time. Nevertheless, all the corresponding deadlock programs are reachable in the directory structure under [28].

[2]https://github.com/vivekShanbhag/Stalemate/ProceduralAnalysis/output/deadlocks/README

Table TC7.3: Types involved in deadlocks: demonstrated violations of lock order

API	Types involved in the deadlock	BugId
Annotation	java.lang.Class(1.1) sun.reflect.annotation.AnnotationType.class(1.5)	6588239
Security (1.5)	sun.security.krb5.internal.rcache.ReplayCache sun.security.krb5.internal.rcache.CacheTable	7118809
MIDI (1.5)	com.sun.media.sound.MidiUtils$TempoCache com.sun.media.sound.RealTimeSequencer	8010771
	com.sun.media.sound.RealTimeSequencer$DataPump com.sun.media.sound.MidiUtils$TempoCache	8010772
	com.sun.media.sound.RealTimeSequencer$DataPump com.sun.media.sound.RealTimeSequencer com.sun.media.sound.MidiUtils$TempoCache	8010774
Corba	org.omg.CORBA.TypeCode.class(1.2) com.sun.org.omg.CORBA.ParameterDescriptionHelper.class(1.3)	JI-9001539
	org.omg.CORBA.TypeCode.class(1.2) com.sun.org.omg.CORBA.ExceptionDescriptionHelper.class(1.3)	JI-9004433
	org.omg.CORBA.TypeCode.class(1.2) com.sun.org.omg.CORBA.AttributeDescriptionHelper.class(1.3)	JI-9013122
	org.omg.CORBA.TypeCode.class(1.2) com.sun.org.omg.CORBA.InitializerHelper.class(1.3)	JI-9013123
	org.omg.CORBA.TypeCode.class(1.2) com.sun.org.omg.CORBA.OperationDescriptionHelper.class(1.3)	JI-9013124
	org.omg.CORBA.TypeCode.class(1.2) com.sun.org.omg.CORBA.ValueMemberHelper.class(1.3)	JI-9013126
	org.omg.CORBA.TypeCode.class(1.2) com.sun.org.omg.CORBA.StructMemberHelper.class(1.3)	JI-9013168
J2SE	java.util.TimeZone.class(1.1) java.io.PrintStream(1.1)	JI-9001540
	sun.util.calendar.ZoneInfo.class(1.4) java.io.PrintStream(1.1)	JI-9004227
	sun.util.calendar.ZoneInfo.class(1.4) java.util.TimeZone.class(1.1) java.io.PrintStream(1.1)	JI-9004229

- it must be multi threaded, with at-least two threads that both share objects (locks) from both these sets.
- the code executed by its threads must be passed such argument values as to enable them to, each, *concurrently* execute code paths that actually realise one of the stack trace options leading to the deadlock.

Using our analysis reports, the following observations help in developing a program whose execution can deadlock its hosting JVM.

- Given a report, one needs access to the Java source of the relevant class-files so as to choose the argument values to be passed to the API to facilitate it to walk the stack as predicted by the report.
- If there exists a program that can indeed deadlock the JVM as is predicted by a particular report, then it is relatively easier to construct such a program if at-least one of the lock sets involved in the deadlock were to be a `Class` object associated with some type. This means that in at least one of the thread stacks, at the top of stack should be a `static synchronized` method. The intuition being that since the `Class` object for a type is unique, and shared by all threads, it lends itself more readily for potential contention.
- If the objects involved in the deadlock are not all created by the library, or if the top of the stack has methods whose implementations have been provided by application types, then it is possibly the user application program that requires correction.
- A Library API usually includes a number of public methods whose invocations do not intend to modify the state of their target object. If such library methods are on top of stack, and the objects involved in the deadlock are also provided by the library, then it is possibly the Library implementation that needs correction.

```
<2-Cycle java/lang/Class sun/reflect/annotation/AnnotationType.class>
  <Thread-1 Option>
    java/lang/Class.initAnnotationsIfNecessary:()V
    .../annotationParser.parseAnnotations:
        ([BLsun/reflect/ConstantPool;Ljava/lang/Class;)Ljava/util/Map;
    .../annotationParser.parseAnnotations2:
        ([BLsun/reflect/ConstantPool;Ljava/lang/Class;)Ljava/util/Map;
    .../AnnotationType.getInstance:(Ljava/lang/Class;)L.../AnnotationType;
  <\Thread-1 Option>
  <Thread-1 Option>...<\Thread-1 Option>
  <Thread-1 Option>...<\Thread-1 Option>
  <Thread-2 Option>
    .../AnnotationType.getInstance:(Ljava/lang/Class;)L.../AnnotationType;
    .../AnnotationType."<init>":(Ljava/lang/Class;)V
    java/lang/Class.isAnnotationPresent:(Ljava/lang/Class;)Z
    java/lang/Class.getAnnotation:(Ljava/lang/Class;)Ljava/lang/annotation/Annotation;
    java/lang/Class.initAnnotationsIfNecessary:()V
  <\Thread-2 Option>
<\2-Cycle java/lang/Class sun/reflect/annotation/AnnotationType.class>
```

Figure FC7.1: A sample deadlock possibility report fetched from JRE analysis.

In genuine cases where the J2SE source needs correction, there are still two cases: either the implementation of offending methods could get modified, resulting in a change in the behaviour of the library, or the usage documentation of the method (Javadoc page contents) could get corrected, better advising users of how to correctly use the functionality offered by the library. Clearly, both cases help to improve the reliability of the Java platform.

7.2 The Annotations case

A lock order violation identified by our analysis, and reported as in Figure FC7.1, prompted us to develop a program[3]: (`Dl.java`) that realises it. For brevity we reproduce only the relevant thread stacks from the report. Notice that among the two objects involved in the deadlock, one would be of type `java.lang.Class`, and the other would be the Class object associated with the type `sun.reflect.Annotation.AnnotationType`.

[3]For more details, see: https://github.com/vivekShanbhag/Stalemate/ApproximateProceduralAnalysis/output/... ...deadlocks/README

As per the report, the two application threads could deadlock if one of them were to call `Class.initAnnotationsIfNecessary()`, and the other call `AnnotationType.getInstance()`, concurrently. Inspecting the Java source file:`java.lang.Class.java`, reveals that its `initAnnotationsIfNecessary()` method is `private`, which applications cannot directly invoke. However, some of its `public` methods, (for instance: `getAnnotations()`) invokes the `private` method *unconditionally*. Notice that, therefore, `Thread1` of `Dl.java`, invokes this method. `Thread2` invokes `AnnotationType.getInstance()` as required by the report. The parameter to these two methods is of type `java.lang.Class`, corresponding to `Test.java`. Notice that `Test.java` needs to be an annotation type, so that it can be a legitimate argument for `AnnotationType.getInstance()`. `Test.java` is from a sun site: [29]. Aspects of the design / implementation of `Dl.java` are inspired from the motivating example. When invoked as in: `javac Test.java Dl.java; java Dl Test`, the JVMs version 1.6.0-beta2-b86, and version 1.5.0_08-b03 do get deadlocked.

"Dl.java" ≡

```
import java.lang.annotation.Annotation;
import sun.reflect.annotation.*;
public class Dl  {
 public static void main(String[] args)  {
  final String c = args[0];
  new Thread() {
   public void run() { try {
     for (Annotation a: Class.forName(c).getAnnotations())
      System.out.println (a);
    } catch (Exception e) { System.out.printf ("T1: %s", e.getCause()); }
   };
  }.start();
  new Thread() {
   public void run() { try {
     System.out.println (AnnotationType.getInstance(Class.forName(c)));
    } catch (Exception e) { System.out.printf ("T2: %s", e.getCause()); }
   };
  }.start();
 }
}
```

"Test.java" ≡

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {}
```

Some executions when the JVM schedules the execution of the two threads such that they do not deadlock, the program produces one of the following two outputs on a successful run to completion.

```
Annotation Type:
   Member types: {}
   Member defaults: {}
   Retention policy: RUNTIME
   Inherited: false
@java.lang.annotation.Retention(value=RUNTIME)
@java.lang.annotation.Target(value=[METHOD])

@java.lang.annotation.Retention(value=RUNTIME)
@java.lang.annotation.Target(value=[METHOD])
Annotation Type:
   Member types: {}
   Member defaults: {}
   Retention policy: RUNTIME
   Inherited: false
```

Java Annotations were introduced in the Version 1.5 of the Java Standard Edition of the platform. In order to support it, new internal classes, including the class called `sun.reflect.annotation.AnnotationType` (among others) were introduced into the distribution. Additionally, more methods were added into existing classes like the `java.lang.Class`. These additional methods included a `synchronized` 'initialisation' method called `initAnnotationsIfNecessary` which would have to invoke methods of the newly added classes, and an enquiry method called `isAnnotationPresent` that the newly added classes would use. This created a reachable violation in the lock ordering that our analysis detects. Had such an analysis been part of the standard Java tool chain, possibly, the cycle would have been detected and removed.

7.3 Three deadlocks in the MIDI API

Our analysis reported three lock order violations in the MIDI (Music Instrument Digital Interface) API, all of which were realised: two of them are 2-cycle, and one is

a three-cycle deadlock. In order to realise these deadlocks, we started with the demonstration programs, freely available from [30]. We made minor changes to one of the sample programs in their distribution, called `RealTimeSequencerTest.java` to fetch the programs[4] that realise the three lock order violations exactly as predicted. It is interesting to note that in some of these deadlocks, one of the threads (of type `com.sun.media.sound.RealTimeSequencer\$PlayThread`) is actually forked off by the library code itself. (Otherwise, in most of the other deadlocks from Table TC7.3 even where the objects and the code involved in the deadlocks belong from the Library implementation, the threads are usually started by the application program).

7.4 Seven deadlocks in the CORBA API

All the 7 deadlock program that realise the 7 predictions are available at: https://github.com/vivekShanbhag/Stalemate/OOAnalysis/output/deadlocks/README.

The version 1.2 of the JRE already contained the Class: `org/omg/CORBA/TypeCode`. JRE version 1.3 introduced all the other classes involved in the 7 CORBA lock order violations (Table TC7.3): these classes, all, have a `static synchronized type()` method whose implementation uses a `synchronized` statement which locks the `java.lang.Class` object corresponding to `TypeCode` class. If an analysis like ours were part of the Java tool chain, these violations would have been detected during the development phase itself, and thereby corrected. All the 7 lock order violations result from the lazy initialisation of their (respective) Class attribute called `__typeCode`. Subsequently, with the introduction of Java memory model from Java 1.5 onwards, one can safely rewrite such circular dependencies along the lines of the "Initialisation on Demand Holder idiom (design pattern)" [31] to remove the lock order violations.

[4]For more details, see: https://github.com/vivekShanbhag/Stalemate/ProceduralAnalysis/output/deadlocks/...
...midi-examples-src-2004-3-31/README

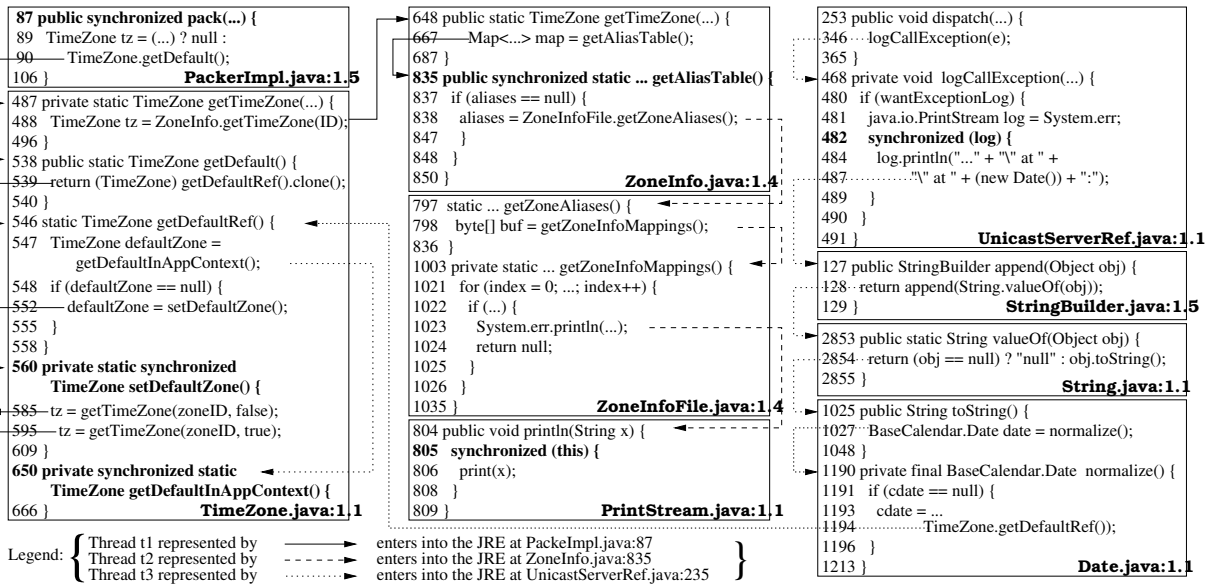

Figure FC7.2: 3-thread 3-lock lock order violation leading to deadlock

7.5 The 3-cycle dynamic dispatch deadlock

As an illustration, below, we discuss an interesting 3-thread 3-lock deadlock program[5] referenced by the last row (bugid: JI-9004229) of Table TC7.3. This program was developed based on the corresponding lock order violation reported by Stalemate.

The code in figure FC7.2 corresponds to the last lock order violation listed in Table TC7.3, which involves 3 threads deadlocked over a set of 3 objects. Each box, in the figure, reproduces relevant source lines of the respective files along with line number information. It depicts the control flow of three threads, named t_1, t_2, and t_3, whose points of entry into the JRE are specified in the legend at the bottom of the picture. t_1 enters the JRE through a `java.util.jar` API, while t_2 enters through the `java.util` API, and t_3 enters via the `sun.rmi` API. Whereas these APIs need not be obviously related, however, the three threads actually deadlock inside the Core Java API! The details of how to execute the program to fetch the predicted deadlocks are in the file:

[5]https://github.com/vivekShanbhag/Stalemate/OOAnalysis/output/deadlocks/TzZiPsDeadlock.java

https://github.com/vivekShanbhag/Stalemate/OOAnalysis/output/deadlocks/README.

The three threads deadlock over the set of three locks, represented by: `(java.util.)TimeZone.class`, `System.err`, and `(java.util.calander.)ZoneInfo.class`. For brevity, we drop the (parenthesised) prefixes of the locks. t_1 locks `TimeZone.class` on line `TimeZone.java:560` and while holding it, attempts locking `ZoneInfo.class` on line `ZoneInfo.java:835`. Concurrently, t_2 locks `ZoneInfo.class` on line `ZoneInfo.java:835` and while holding it, attempts locking `System.err` on line `PrintStream.java:805`. Concurrently, t_3 locks `System.err` on line `UnicastServerRef.java:482` and while holding it, attempts locking `TimeZone.class` on line `TimeZone.java:650`.

Notice, at the right hand bottom of every box, the name of the file and the JRE version into which it was first included are mentioned. Thread t_1 enters into the library by invoking the method of a Class from 1.5, which invokes methods of Classes from 1.1. One of these then invokes into a Class from 1.4! Similarly, along the stack for thread t_2, a method from some Class of 1.1 invokes into a Class method from 1.5! Since the versions 1.4 & 1.5 of the JRE came years later than its version 1.1, such control flow ought to be surprising. Occasionally one may attribute it to the many bug fixes that keep getting into the update releases that cause such turmoil in the code. Alternatively, the possibility that dynamic dispatch causes such control flow may be cited. However, we find that 480 out of 714 lock order violations (67%) reported by Stalemate result from such control flow. We believe that if such an analysis had been accessible to the designers of the JRE as a component of the Java Language tool chain, all along, then in many cases such control flow would not have been released to the user community.

The above lock order violation is also interesting for another reason: This is a case of deadlock due to dynamic method dispatch. While `logCallException` invokes `append` handing it a `Date` object as parameter, the `javac` compiler resolves it against the `StringBuilder.append:(LObject;)` method definition. The `append` method

then invokes `String.valueOf()` handing it the `Date` object that it received. The `String.valueOf()` method invokes the `toString()` method on its argument, that the JVM would dispatch to `Date.toString()` method which overrides the `Object.toString()` method that the compiler resolved it against. Our analysis is able to predict such dynamic dispatch accurately. Conventional static analysis that relies upon the declared types of variables/parameters would not be able to perform so.

CHAPTER 8

AVOIDING DEADLOCKS USING STALEMATE AND DIMMUNIX

We have already seen that static analysis approaches can predict deadlocks in Java libraries [8, 23] and programs [10]. Our prototype implementation Stalemate [28] predicts sets of reachable call stacks that may deadlock if realised concurrently. Dimmunix [12] is a tool that actively prevents *known* deadlocks from re-manifesting. Dimmunix must witness a program execution that leads to deadlock. In the process it records the *deadlock fingerprint* which is read and used in subsequent executions to avoid *that* deadlock. So that it can be used during subsequent executions, *dynamic* aspects of the deadlocked state are abstracted away when recording the *fingerprint*. It turns out, the format and information content of the Dimmunix fingerprint is such that it can be generated through static analysis. In this chapter we use the lock order violations predicted by Stalemate [28], as well as 8 of the 15 programs whose execution can deadlock as predicted. We explore the possibility of automatically rewriting the lock order violations as deadlock fingerprints for Dimmunix to use directly. Generating fingerprints through static analysis is a way to forewarn Dimmunix of all deadlock possibilities rather than it have to *learn* about them, one at a time until it is completely *vaccinated*. This makes it unnecessary for the application to ever deadlock.

Dimmunix: This tool [32] comprises of two modes of operation: observation mode, and deadlock avoidance mode. In order to allow Dimmunix to perform its role in either of these two modes, it must be allowed to *witness* the executions of the target program. In its 'observation' mode, Dimmunix is not already aware of the deadlock that the target program is about to get itself into. Whenever any such *new* deadlock is reached, Dimmunix records the *fingerprint* of the deadlock. During subsequent executions of the same program, since Dimmunix is able to fetch the fingerprints of all the deadlocks that it has already witnessed from the history that it saved during earlier deadlocked executions, it avoids them by preventing the JVM from scheduling the participating threads in any manner wherefrom the respective deadlocks become reachable. In order to implement this strategy, Dimmunix uses bytecode transformation to rewrite portions of participating method bodies during class loading.

The Java prototype for Dimmunix comprises of a Java agent, and the above described *deadlock avoidance strategy (das)*. The Java agent installs a bytecode transformer which employs the instrumentation API to engineer bytecode of classes loaded during application execution, inserting calls to *das* at appropriate points: before and after every `monitorenter` and also before every `monitorexit`. The *das* maintains a *Resource Allocation Graph (RAG)* whose nodes represent locks & threads and directed edges represent the acquired / requested relations amongst them. If the execution reaches an unknown deadlock, the RAG is used to construct the corresponding deadlock fingerprint that is recorded in a history file. Subsequent executions read the fingerprints, and based on how the RAG evolves, *das* may block the acquisition of an (otherwise) available lock by some thread, to avoid a known deadlock.

Stalemate: Recollect that lock order violations predicted by this tool [28] are of the form $< T_1, ...T_i, ...T_k > \equiv Ts_1, \ ...Ts_i, ...Ts_k$, where $T_i, \forall 1..k$ are types. In general, Ts_i is a set of possible thread stack fragments along each of which (at least) two locks are acquired in a nested fashion: the first being an object of type T_i, and the last of

type $T_{(i+1) \ mod \ k}$. A 2-cycle prediction is interpreted as: given an application whose one thread realises a member of Ts_1 and another thread realises a member of Ts_2 concurrently and if they share the locks, then they could deadlock. Stalemate publishes 15 Java programs whose executions can deadlock as predicted by the static analysis of the JRE. For the investigation described in this chapter we use 8 of these programs.

8.1 Objective of the Investigation

Figure 8.1 illustrates the organisation of the tools that the investigation centres upon. The bounding box '1' encapsulates the two essential pieces of input to our investigation that are inherited from the outcome of Stalemate. The analysis of the JRE produced the lock order violations. Based upon some of them the deadlocking programs were manually developed. The bounding box '2' comprises of Dimmunix being witness to the execution of deadlocking programs in both its operating modes. The box '2.1' depicts the 'observer' mode of Dimmunix operation that leads to the populating of the 'fingerprints history file'. Box '2.2' depicts the 'deadlock avoidance' mode of Dimmunix operation which reads and uses the deadlock fingerprints from the history file to avoid 'known' deadlocks. However, when any new deadlock is reached, whose fingerprint is not yet present in the history file, Dimmunix in its 'observer' mode records the new fingerprint. The two modes of operation of Dimmunix co-exist. The objective of our investigation is: whether the lock order violations fetched from Stalemate may be automatically re-written as Dimmunix fingerprints into the history file so as to unnecessitate its observer more of operation that requires the program to deadlock, once, in every possible way. Notice, in figure 8.1, the edge that asks: 'Is rewriting possible'?

We would like to use some of the deadlock programs, that are available form the Stalemate effort, and arrange for Dimmunix to observe their execution. When the execution reaches a deadlock, Dimmunix would generate the fingerprint for it. Subsequent

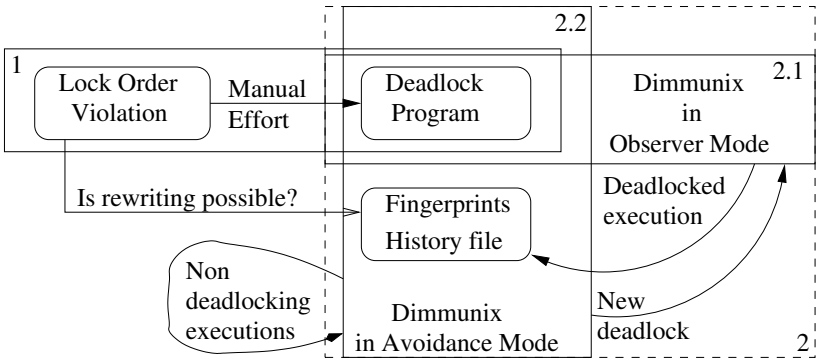

Figure FC8.1: Organisation of the tools subject to Investigation

executions of the same program conducted similarly under the supervision of Dimmunix ought not to deadlock in the same way: this must be ascertained. Finally, we would like to compare this fingerprint with the corresponding lock order violation to arrive at a conclusion as to whether or not the rewriting may be automated.

However, since not all executions of deadlocking programs may deadlock, the above approach poses two problems:

- In our attempt to get Dimmunix to generate the fingerprint for the deadlocks, what is to be done if it takes arbitrarily long time for the first deadlocking execution to occur?
- (Assuming we are lucky enough to succeed in fetching the fingerprint for a deadlock) When ascertaining that Dimmunix is able to use the fingerprint to avoid similar deadlock, how can we be sure that some execution would indeed have proceeded to deadlock had it not been avoided by Dimmunix, equipped with the fingerprint?

In order to address both the above concerns, we develop an environment using a clever composition of some standard tools available on Unix such that every execution of a deadlocking program within it deadlocks exactly as predicted. This environment is not generic: it is not applicable for arbitrary deadlock programs. Nevertheless, for the

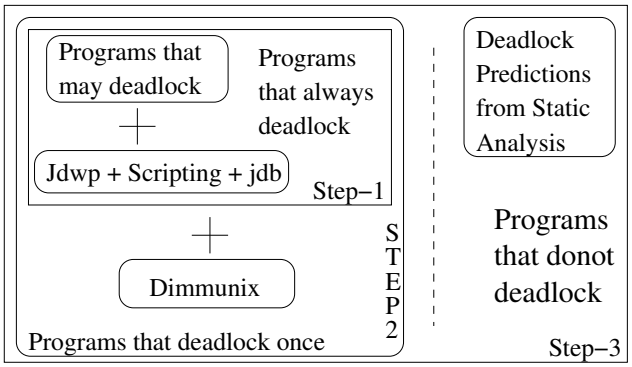

Figure FC8.2: The Step-wise Method of Investigation

deadlock programs available from Stalemate [28], which deadlock exactly as predicted by the corresponding lock order violations, the environment we have composed, does the job.

Starting with 8 programs whose executions sometimes deadlock, our three step approach (see figure 8.2) proceeds as follows: In step 1 we confirm that the program execution within our environment does indeed always lead to deadlock. In step 2 we incorporate Dimmunix into our environment and verify that when program execution does deadlock, Dimmunix records the deadlock fingerprint and also that all subsequent executions do not similarly deadlock. Step 3 describes the Dimmunix fingerprint structure, and how Stalemate may generate deadlock fingerprints for Dimmunix so deadlock is avoided in all executions of the programs.

8.2 Always deadlocking a deadlock program

Given a Java program whose execution sometimes deadlocks, we want to deadlock its every execution. In general, during the execution of an n-thread n-lock deadlock program, if the JVM interleaves the execution of its n threads such that each of them acquires one lock, then all such executions will proceed to deadlock. Recollect that the 8 programs' executions deadlock as predicted by their respective lock order violations.

Given a program whose execution deadlocks, and the corresponding lock order violation, we identify execution points from the stack prediction for (its) every program thread, where it would have acquired the first lock but not (yet) the second. During its execution, at such points for all threads, we suspend their execution and `yield` control to one of the other threads. Once the execution of all (participating) program threads has so been suspended, (after they have acquired their first lock, but not (yet) the second) we resume execution for all of them. Irrespective of how inter-leaving / execution of the threads proceeds from this point onwards, all executions would lead to deadlock.

In order for the JVM to be able to exercise sufficiently fine level of control over thread scheduling, so as to allow only such inter-leavings, we choose `jdb` (over the conventional `java`) to oversee program execution. `jdb` allows program execution to be stopped at breakpoints and facilitates the program state be be witnessed. Moreover, program threads can be suspended and resumed. Using such features of `jdb`, we have developed a script-assisted execution environment that enforces the above strategy. By restricting the choice of thread-inter-leavings upon program execution in this manner we ensure that every execution within this environment does lead to deadlock.

Consider one of the deadlock programs, `TcEdh.java`, whose execution can deadlock as predicted by the lock order violation reproduced below.

"TcEdh.java" ≡

```
public class TcEdh {
  public static void main(String[] args) {
    new Thread() {
      public void run() { com.sun.org.omg.CORBA.OperationDescriptionHelper.type(); };
    }.start();
    new Thread() {
      public void run() { com.sun.org.omg.CORBA.ExceptionDescriptionHelper.type(); };
    }.start();
  }
}
```

Lock Order Violation that predicts the deadlock of TcEdh.java ≡

```
<Cycle-2 locks=" ooC.TypeCode.class csooC.EDH.class">
```

```
<Thread-1>
<csooC.ODH.class>csooC.ODH.type:()LooC/TypeCode;
<ooC.TypeCode.class>csooC.ODH.type:()LooC/TypeCode;#LooC/TypeCode.class;#0
  =PartOf=csooC.ODH.type:()LooC/TypeCode;
<csooC.EDH.class>csooC.EDH.type:()LooC/TypeCode;
</Thread-1>
<Thread-2>
csooC.EDH.insert:(LooC/Any;LcsooC/ExceptionDescription;)V
<csooC.EDH.class>csooC.EDH.type:()LooC/TypeCode;
<ooC.TypeCode.class>csooC.EDH.type:()LooC/TypeCode;#LooC/TypeCode.class;#0
  =PartOf=csooC.EDH.type:()LooC/TypeCode;
</Thread-2>
</Cycle-2>
```

[Note: In the lock order violation, for brevity, we use the abbreviations: 'ooc' is short for 'org.omg.CORBA', and 'csooC' is short for 'com.sun.org.omg.CORBA'. Similarly, 'EDH' is short for 'ExceptionDescriptionHelper', and 'ODH' is short for 'OperationDescriptionHelper'.] Notice the correspondence between the respective parts of the Java program, and the lock order violation above: the body for the first thread of the program invokes the entry point for the `Thread-1` stack prediction, and the body for the second thread of the program invokes the entry point for the `Thread-2` stack prediction. Every line in the stack prediction corresponds to the behaviour of a code fragment that is either an entire method body or a `synchronized` statement in some method. Where the code fragment acquires a lock, the type of the lock is reproduced as the prefix of the line, encapsulated by the $< \& >$ symbols.

Execution of the program `TcEdh` creates two threads, and their execution may interleave in many different ways. At the level of instruction executions, there are very many feasible inter-leavings of the two threads. However, the inter-leavings can all be classified into four classes of schedules: two of which lead to deadlock, and the others do not lead to deadlock. The four classes are described below:

- The first thread's execution succeeds in acquiring the lock on `TypeCode.class` before the second thread can request its first lock. Subsequently, however, the second thread succeeds in acquiring the lock on `ExceptionDescriptionHelper.class`

before the first thread gets to the point of requesting it. All schedules that execute in this manner do proceed to deadlock.

- The second thread's execution succeeds in acquiring the lock on `ExceptionDescriptionHelper.class` before the first thread can request the lock on `TypeCode.class`. Subsequently, however, the first thread succeeds in acquiring the lock on `TypeCode.class` before the second thread gets to the point of requesting it. All schedules that execute in this manner do proceed to deadlock.

- The first thread's execution succeeds in acquiring both the locks involved in the deadlock: `TypeCode.class` as well as `ExceptionDescriptionHelper.class`, before the second thread can request its first lock. All schedules that execute in this manner do not lead to deadlock.

- The second thread's execution succeeds in acquiring both the locks: `ExceptionDescriptionHelper.class`, as well as `TypeCode.class`, before the first thread can request the lock: `TypeCode.class`. All schedules that execute in this manner do not lead to deadlock.

In general, therefore, once all participating threads successfully acquire one of the participating locks, deadlock is certain. Such a scheduling of the participating threads can be imposed upon the JVM. The java debugger, `jdb` makes it so possible. Below, we illustrate how this can be done for the execution of `TcEdh`.

The terminal session of deadlocking TcEdh execution: (reproduced) ≡

```
$ jdb TcEdh
Initializing jdb ...
> stop at csooC.OperationDescriptionHelper:46
Deferring breakpoint csooC.OperationDescriptionHelper:46.
It will be set after the class is loaded.
> stop in csooC.ExceptionDescriptionHelper.type
Deferring breakpoint csooC.ExceptionDescriptionHelper.type.
It will be set after the class is loaded.
> run
run TcEdh
Set uncaught java.lang.Throwable
```

```
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint csooC.OperationDescriptionHelper:46
Set deferred breakpoint csooC.ExceptionDescriptionHelper.type

Breakpoint hit: "thread=Thread-1", csooC.ExceptionDescriptionHelper.type(), line=42 bci=0

Thread-1[1] threads
[...]
  (TcEdh$1)0x14c                                Thread-0          running
  (TcEdh$2)0x14e                                Thread-1          running (at breakpoint)
  (java.lang.Thread)0x14d                       DestroyJavaVM     running
Thread-1[1] suspend 0x14e
Thread-1[1] cont
>
Breakpoint hit: "thread=Thread-0", csooC.OperationDescriptionHelper.type(), line=46 bci=12

Thread-0[1] resume 0x14e
Thread-0[1] cont
>
```

The session starts with the command to execute the program: `jdb TcEdh`. Once program execution is initialised by jdb, it awaits user instruction. At this point the two breakpoints are set: the first one is at a code-point after the first thread acquires the first participating lock using a `synchronized` statement, and the second is after entry to the `synchronized` method that the second thread invokes to acquire its first participating lock. Notice in the terminal session, later, use of the jdb command `threads` to fetch the information about the thread-id that must be `suspended`. The suspended thread would have already acquired its first participating lock. The other break-point is then reached: signifying that other participating thread has acquired its first participating lock. At this point all suspended threads are `resumed` and the program execution `continued`. The execution proceeds to deadlock.

We have implemented a generalised version of the above strategy in the form of a little Java program (called `Ads.java`, for *Always deadlocking strategy*) that administrates it for n-thread n-lock programs that can deadlock as predicted. The input for `Ads` is the name of the deadlock program, and the various breakpoints at which its each participating thread would have acquired the first of its participating locks but not yet the second.

jdb is used to oversee the execution of the deadlock program, as described above. The two-way communication between Ads and jdb is conducted over two Unix pipes. Our always deadlocking execution environment, therefore, comprises of the above components. Using this execution environment we are able to ensure that all executions of each of the 8 programs do indeed deadlock as predicted.

8.3 Dimmunix + Always Deadlocking Strategy

We want Dimmunix to witness the execution of these programs, as conducted by Ads. We would like to verify, for a given deadlock program, that its first such execution does deadlock, and consequently Dimmunix is able to fetch the corresponding deadlock fingerprint. Furthermore, we would like to ascertain that subsequent such executions of the program do not deadlock since Dimmunix is able to avoid it. Unfortunately, we were unable to integrate Dimmunix into the execution environment exactly as described in the previous section. Nevertheless, with appropriate modifications to the execution environment we are able to accommodate Dimmunix, as desired.

We could not get Dimmunix to witness program execution that is conducted, in-process, by jdb. However Dimmunix is able to witness the program execution when it is conducted by java, and there is a way that jdb can be made to attach to such a java process. Therefore, we choose to make a slight modification in our execution environment: Instead of using jdb in-process, we opted to attach it to the program execution that is conducted by java, and witnessed by Dimmunix. Figure 8.3 illustrates the upgraded execution environment which comprises of three processes: Ads continues to communicate with jdb using the two Unix pipes, and on the other side, jdb communicates with java over a Tcp/IP Socket.

Dimmunix in process co-exists with java in the JVM. The four components to the left of the dotted vertical line in the JVM box are the notable aspects of Dimmunix: The

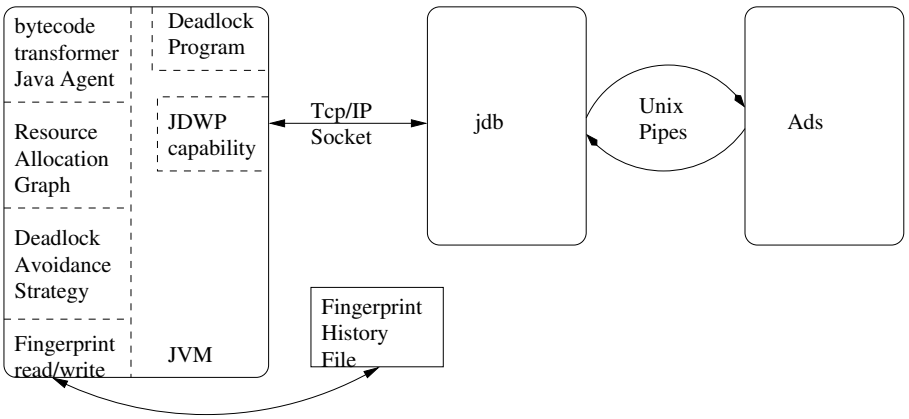

Figure FC8.3: Importing Dimmunix into Always Deadlocking Execution Environment

'bytecode transformer Java Agent' is the component that rewrites bytecode during class loading to ensure that its 'deadlock avoidance strategy' is notified when locks are acquired or released. The 'deadlock avoidance strategy' updates the 'resource allocation graph (RAG)' that represents locks and threads in the application space as nodes of the RAG. Whenever it notices that granting a request for an available lock would close a cycle in the RAG, it *hold*s the request until the RAG is updated as a result of the release of other conflicting locks held in other concurrently executing threads. Subsequently, once it realises that the cycle will not close, anymore, it releases the *hold*, resulting in granting the requested lock. The last component is the one that generates and reads the fingerprints in the 'Dimmunix history file'.

For our example program: `TcEdh.java`, the above environment is used as follows:

1. **Setup**: Create two Unix pipes be executing the commandline:
   '`mkfifo /tmp/readFIFO /tmp/writeFIFO`'. Also open three terminal sessions, one each to initiate the execution of the three commands below.

2. **java**: In the first terminal, start the JVM using the commandline:

```
java -Xdebug  -classpath Dimmunix/java/test \\
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=localhost:1044 \\
-Xbootclasspath/p:Dimmunix/java/src/Dimmunix/bin:./asm-3.2/lib/asm-3.2.jar \\
-javaagent:Dimmunix/java/src/DimmunixInstrumentation/DimmunixAgent.jar \\
dimmunixTests.TcEdh
```

3. **jdb**: In the second terminal start `jdb` using the commandline:

```
script -f -c 'cat /tmp/writeFIFO | jdb -attach localhost:1044' /tmp/readFIFO
```

4. **Ads**: In the third terminal start `Ads` using the commandline:

```
java Ads com.sun.org.omg.CORBA.OperationDescriptionHelper:46 \\
         com.sun.org.omg.CORBA.ExceptionDescriptionHelper.type
```

When Dimmunix is made witness the first execution of `TcEdh` as above, its deadlock detection and deadlock fingerprint recording capability is triggered, resulting in an update to the history file. Subsequent similar executions of `TcEdh` do not deadlock since Dimmunix avoids it. We are able to verify / demonstrate both these modes of operation of Dimmunix in our execution environment.

8.4 Fingerprints from Lock Order Violations

Recollect the objective of our investigation: Is it possible to generate the deadlock fingerprint, given the deadlocking program and its corresponding deadlock prediction fetched from static analysis? The answer is 'Yes'. Details follow:

Lock order violations from Stalemate: Lock order violations reported by Stalemate consist of sets of stack traces. Each stack trace is a *hyphen separated* sequence of code fragment identifiers. The code fragment may be either an entire method body or just a `synchronized` statement. In either case, the code fragment identifier consists of the method signature for the method. In this way, Stalemate uses method signatures to identify execution points in the code. On the other hand, Dimmunix fingerprints use the *ClassNm.MethodNm (fileNm:LineNum)* format.

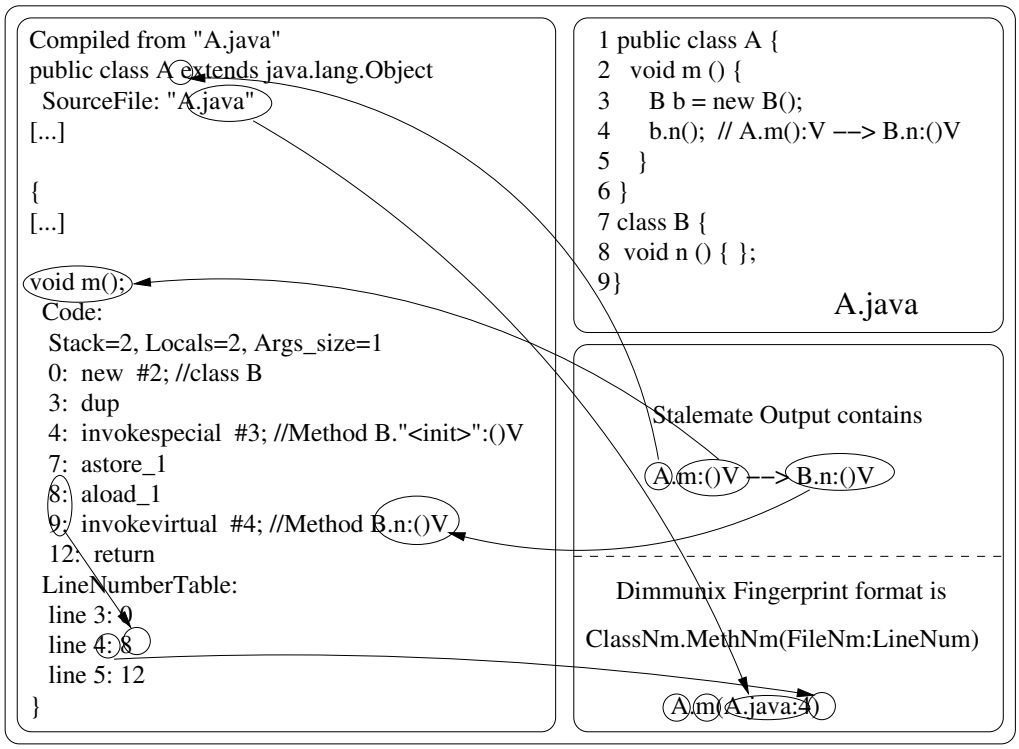

Figure FC8.4: Fetching Dimmunix fingerprint from Lock Order Violations

Dimmunix fingerprints: The Dimmunix history file records one deadlock fingerprint per line, for every uniquely reachable deadlock witnessed. For an *n*-thread deadlock, the fingerprint comprises of a *semicolon separated* serialisation of the fragments of *participating thread* stacks that lead up to the acquisition of the first *participating lock* for that thread. The serialisation of the thread stack is a *comma separated* list of stack trace elements. The *stack trace element* format, as mentioned above, is identical to that in the output of the JVM's deadlock detection component. Re-writing: Figure 8.4 illustrates, pictorially, how code positions in the Dimmunix format can be fetched from the Stalemate format, using the information from the Class files. The top right portion of the figure reproduces the contents of file `A.java`, including line numbers, that defines classes, `A`, and `B`. On its line 4, the method `A.m()` invokes method `B.n()`. `A.java` is compiled and the generated class file A.class is disassembled, using commandline: '`javac A.java; javap -verbose A`'. The left hand side column in figure 8.4 contains positions of the output of `javap`. The bottom right of the figure has two portions:

the upper portion depicts the Stalemate format of identifying the code position on line 4 of `A.java`, and the lower portion illustrates the same code position in the Dimmunix format. The Arrows from the upper portion to the left hand side, and from there to the lower portion illustrate the re-writing method. In particular, looking up the byte-code index of the `invoke`eation of `B.n()`, in the `LineNumberTable` corresponding to the method body for `A.m()` fetches the required *LineNum*. The class header for `A`.class contains the *fileNm*. *ClassNm.MethodNm* is `A.m`.

8.5 Results

The results of our investigation, using the 8 deadlock programs, are tabulated in the Table TC8.1. All the 8 deadlock programs passed Step 1: we succeeded in 'always deadlocking' their execution. Six of them passed Step 2: Dimmunix works as expected in both modes. The manual step 3 of verifying the rewriting of stack traces using the 'LineNumber Table(s)' also succeeds for these 6 programs.

The Annotation & Security API deadlocks fail step 2 because Dimmunix is unable to instrument `java.lang.Class` and `java.util.Hashtable`, respectively. Adding deadlock fingerprints into the Dimmunix history file based upon the corresponding predictions also does not help it to avoid the deadlocks: Since those classes are not instrumented, their methods acquire and release locks without updating the RAG, and bypassing the *das* component of Dimmunix. More than 425 JRE-classes (`java.lang.Class` & `java.util.Hashtable` included) are loaded by the JVM even before the Java agent is *preloaded*. As a result, deadlocks involving their methods cannot be avoided by this approach.

Table TC8.1: Stalemte + Dimmunix: Results of the Investigation

API	Program	BugId	step1	step2	step3
MIDI:	RtsTc	8010771	Pass	Pass	Pass
Musical	TcDp1	8031702	Pass	Pass	Pass
Instrument	TcDp2	8031698	Pass	Pass	Pass
Digital I/f	RtsTcDp	8031699	Pass	Pass	Pass
Corba	TcPdh	8024334	Pass	Pass	Pass
	TcEdh	Awaited	Pass	Pass	Pass
Annotation	JlcAt	6588239	Pass	Fail	Fail
Security	RcCt	7118809	Pass	Fail	Fail

CHAPTER 9

CONCLUSIONS

We provide summary of our contribution and the future work later in the chapter. The 1,85,000 cycles reported by our analysis of JRE version 1.6 is indeed quite large. However, we would like to observe the following:

- Every one of them identifies a set of call stack possibilities that do imply a lock order violation. In a life/mission critical context, perhaps, not a single such violation is permitted. However, the JRE is predominantly used in the commercial space, where users, customers and providers of software are all somehow more willing to put-up with outages.
- Nevertheless, for instance, in all the CORBA, MIDI & Security API lock order violations we have realised, leading to deadlock the execution of the corresponding program, the violations involved only one cycle each: with one forward stack, and one reverse stack, and they were both realisable concurrently while sharing objects. When the realisability of a lock order violation can so readily be demonstrated, it certainly is a big help to the library maintenance team.
- Since a tool like Stalemate has not been available to the library developer such lock order violations have been accumulating over the years, since the time JRE 1.1 was first made available. For instance, the release-wise breakup of the 100 sets of violations that involve classes from the same version of the API is as follows:

1.1(86), 1.2(10), 1.3(1) and 1.5(3). The other 614 out of 714 sets of violation involve classes across releases of the JRE.

9.1 Invocation Graph *vs* Context Sensitive Call Graph

In the literature [14,16], the usage "context sensitive call graph" refers to call graphs constructed using an underlying alias analysis approach that models collections of heap objects allocated at the same allocation site by a single abstract symbolic heap object. Our invocation graph differs from this approach: we use a single object to represent all objects of the same type, irrespective of the allocation site they are allocated at.

Another notable difference between the two graphs is that in most casess the 'contest sensitive call graph' is used to represent a call tree corresponding to a whole program. Whereas, our work uses the invocation graph to store, collectively, the call graphs emanating (downwards) from potentially every entry-point into the library (from application code).

The distinctions above lead to a significant reduction in the space required to store the invocation graph as compared to the context sensitive call graph. Furthermore, this thesis gives a constructive account of how, starting from the output of the compilation phase, the analysis builds the invocation graph for the entire Java Standard Library; this, we believe has not be done before.

9.2 Comparison With Other Approaches

It is interesting to compare the results fetched from our analysis with those reported from other approaches. Since ours is a static analysis approach, we start by comparing with other static approaches.

Our analysis reports a very large set of lock order violations for the JRE. Certainly, a significant proportion of them may not be realisable due to various runtime conditions. Moreover, specifically since we choose not to permit self loops in our *tlog*, the deadlocks like from [33] are not located by our analysis. Whereas, other static analysis methods including [8–10] are able to flag such deadlocks.

On the other hand, ours is the first work that reports finding of three-cycle deadlocks (one in the MIDI API, and another in the Core Java API) based upon analysis by Stalemate. All other reports look for only 2-cycle deadlocks. One of the call stacks in the Core Java 3-Cycle lock order violation involves a dynamic method dispatch prediction that is noteworthy.

In general, typically, other static analysis approaches propagate the declared return types from method invocations. Whereas, our thesis motivates the need to propagate the actual type of the object returned from an method invocation; We also formalise how this can be done. To the best of our knowledge, this has not been attempted before.

Limitations of Static Analysis Methods: Nevertheless, not withstanding any of the discussions above, it is indeed possible to develop a simple program using the Reflection API whose purpose is to deadlock its very execution. Based upon the lock order violations fetched from our analysis of the JRE, execution of the following program can demonstrate this.

"CorbaDeadlock.java" ≡

```
import java.lang.*;
import java.lang.reflect.*;
public class CorbaDeadlock {
  public static void main(String[] args) {
    assert (args.length == 2);
    try {
      final Method m1 = Class.forName(args[0]).getDeclaredMethod("type", (Class[]) null);
      final Method m2 = Class.forName(args[1]).getDeclaredMethod("type", (Class[]) null);
      new Thread() {
        public void run() { try {m1.invoke(null, (Object[]) null);} catch (Exception e){} };
      }.start();
      new Thread() {
        public void run() { try {m2.invoke(null, (Object[]) null);} catch (Exception e){} };
```

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

No static analysis of `CorbaDeadlock`, above, can identify any lock order violation in it nor detect the possibility of a deadlock during its execution. However, both its executions, as below, can lead to deadlock:

- Initiate its execution using the command line:

```
java -ea CorbaDeadlock com.sun.org.omg.CORBA.ParameterDescriptionHelper \\
                       com.sun.org.omg.CORBA.OperationDescriptionHelper
```

- Alternatively, initiate its execution using the command line:

```
java -ea CorbaDeadlock com.sun.org.omg.CORBA.ExceptionDescriptionHelper \\
                       com.sun.org.omg.CORBA.OperationDescriptionHelper
```

The expressiveness of the Java language using its various APIs is indeed very powerful. Interestingly, approaches based upon dynamic analysis methods, including [12, 19] are able to handle both the above deadlocks. For instance, Dimmunix can generate the signatures for both the deadlocks reached as above, and subsequent executions of the program would not deadlock.

9.3 Contributions, Limitations & Future Work

Our original intent to design a global, OO, type based static analysis that honours all the language constructs and may be bundled into a tool component to integrate into the Java programming language tool chain has, in the most part, been met. Certain limitations in our approach and our tool are discussed later in this section.

There are the two aspects of an OO library: (1) that it is designed to offer types (Classes and Interfaces) to the application programmers that they find useful in accomplishing *their* task; (2) that such a bunch of types are to be implemented using the support of utility classes and whatever support system is available. Generally the API designer takes the first view, and the library programmer takes the second view; The two views are not mutually exclusive. While the design of our analysis is very generic, in that it honours all the Language features and there are no handicaps built into it for any reason. However, certain design choices about how we apply it for studying the JRE are taken so that its results become useful for the library programmer (rather than the API designer).

In the detailed discussion regarding abstract interpretation, we visualise two options when it comes to choosing the implementation strategy: either a quick but imprecise option, or a precise but elaborate option. Our implementation [28] has been crafted using the quick but imprecise option. However it can be upgraded to provide the other option without too much effort. The other approach necessitates a fixed point computation which can maximise precision.

The 15 deadlock programs that we have developed starting from the lock order violations predicted by the analysis establish, beyond reasonable doubt, the accuracy of the thread stacks fetched from the analysis. The thesis also discusses the intuition: how to develop the deadlock program given a lock order violation.

From our exploratory work using Dimmunix, we find that it explores an effective approach to deal with known deadlocks in applications. We are able to demonstrate how its usefulness can be enhanced by providing it fingerprints fetched from static analysis. The two failure cases in table 8.4 expose the subtle limitations of the engineering approach employed by the current implementation of Dimmunix.

Present Limitations: Our work described in this thesis has the following limitations

in its approach:

- We discount unnamed initialisers of classes and instancees. Among these, static initialisers are invoked after successful class resolution, and instance initialisers are invoked just before the constructor. Our analysis currently does not model these. Cycles reachable through such blocks of code are therefore not detected or reported.

- We don't take cognisance of the definitions of the 1927 `native` methods defined as a part of the JRE. 1927 out of 160000 is a relatively small fraction (0.0120125%). However, 55 of them are `synchronized` ! The total number of `synchronized` (statements, as well as instance and class methods) code we model is 6296. Effectively we are ignoring at least 55/6351 (.866%) of the definitions that gives rise to the problem.

- Recollect that to contain the number of cycles reported, we used a constrained definition for *tlog*-edge. The total number of lock order violations in the JRE are more than what our analysis currently fetches.

- In our *tlog*, we disallow self edges. Therefore deadlocks on two objects of the same class remain undetected. A good example of such a deadlocks, that we cannot predict is, in Listing 10.2, of pg. 208, [33].

- The analysis described in this thesis, and as implemented by the prototype tool Stalemate is a simple text book like approach. To discard unrealisable call stacks one can employ algorithms like, thread escape analysis, and many others.

Future Work: This thesis explores and reports upon the use of the described approach to analyse large libraries (like the JRE). The design & implementation of Stalemate needs to be upgraded so that it may carry out a similar analysis for the classes comprising an application, exploring also the reachable subset of the JRE, therefrom.

With a view to reduce the “false positives”, so that the programmer is motivated to attach greater significance to the findings of the analysis, further work is necessary to identify call stacks that need not be flagged and then to upgrade our method to weed them out from the output. The “thread escape analysis” used in [10] is a good example of a method that can help in this direction.

In Chapter 2 we observed that other static analysis approaches based upon alias analysis tend to fetch very large models of the input. As a result of our analysis, based upon types, we find that only about 10% of the classes defined in the JRE are primary contributors to deadlock possibilities. A two tiered approach that starts by employing a type based analysis like ours, followed by a more detailed selective modelling using alias analysis methods may likely fetch a smaller model of the input, but at a finer level of granularity that is effective in reducing the false positives.

It will be interesting to examine the findings from applying the present analysis to the C# Langauage Library. We have conducted some initial investigation [34], and the study confirms its applicability to C#/MSIL.

Exploring the scalability of the Dimmunix approach to avoid deadlocks reachable using exploits predictable through static analysis of legacy libraries (like JRE) can be useful. This makes it unnecessary to continue viewing them as bugs that require fixing.

Combining our analysis with the work described in [20] can lead to interesting results. Their synthesis of multi-threaded programs requires a set of stack traces as a starting point. Currently they fetch *realisable* stack traces from witnessing executions of sequential test programs from the regression test suite, for instance. The effectiveness of their approach is therefore quite sensitive to the coverage of the test suite they use. On the contrary, an analysis like ours can give them all *reachable* sets stack traces that may *potentially* cause lock order violations. Automating the synthesis of programs that may realise members of such sets of stack traces to cause deadlock would be wonderful.

Bibliography

[1] Brian W Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[2] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997.

[3] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[4] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[6] Jeremy Manson and Brian Goetz. Jsr 133 (java memory model) faq. `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`.

[7] Frank Yellin and Tim Lindholm. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[8] Amy Williams, William Thies, and MichaelD. Ernst. Static deadlock detection for java libraries. In AndrewP. Black, editor, *ECOOP 2005 - Object-Oriented*

Programming, volume 3586 of *Lecture Notes in Computer Science*, pages 602–629. Springer Berlin Heidelberg, 2005.

[9] Jyotirmoy Deshmukh, E Allen Emerson, and Sriram Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 480–491. IEEE Computer Society, 2009.

[10] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, pages 386–396. IEEE Computer Society, 2009.

[11] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Pasareanu, Hongjun Zheng, et al. Bandera: Extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448. IEEE, 2000.

[12] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George C. Deadlock immunity: Enabling systems to defend against deadlocks, 2008.

[13] Fancong Zeng. Pattern-driven deadlock avoidance. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '09, pages 6:1–6:8, New York, NY, USA, 2009. ACM.

[14] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM.

[15] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proceedings of the 13th Australian Conference on Software Engineering*, ASWEC '01, pages 68–, Washington, DC, USA, 2001. IEEE Computer Society.

[16] Ondej Lhotk and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, pages 47–64. Springer Berlin Heidelberg, 2006.

[17] Fancong Zeng. Deadlock resolution via exceptions for dependable java applications. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 0:731, 2003.

[18] Fancong Zeng and Richard P Martin. Ghost locks: Deadlock prevention for java. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.

[19] Malavika Samak and Murali Krishna Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2014.

[20] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 473–489. ACM, 2014.

[21] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 327–336, New York, NY, USA, 2010. ACM.

[22] NASA Ames Research Centre. Jpf: The swiss army knife of java verification. `http://babelfish.arc.nasa.gov/trac/jpf`.

[23] Vivek K. Shanbhag. Deadlock-detection in java-library using static-analysis. *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, 0:361–368, 2008.

[24] Christoph von Praun. *Detecting synchronization defects in multi-threaded object-oriented programs*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[25] Surabhi Pandey, Sushanth Bhat, and Vivek Shanbhag. Avoiding deadlocks using stalemate and dimmunix. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 602–603. ACM, 2014.

[26] Xavier Leroy, Inria Rocquencourt, and Trusted Logic S. A. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 2003.

[27] Allen Goldberg Kestrel and Allen Goldberg. A specification of java loading and bytecode verification, 1998.

[28] Vivek. K. Shanbhag. Static analysis of the entire jre to predict deadlock exploits. `https://github.com/vivekShanbhag/Stalemate`.

[29] Sun Microsystems J2SE Team. Sun microsystems: Java annotations tutorial page. `http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`.

[30] Florian Bomers and Matthias Pfisterer. jsresources.org: Java sound resources. `http://jsresources.org`.

[31] Wikipedia. Initialization-on-demand holder idiom. `http://en.wikipedia.org/wiki/Initialization-on-demand\_holder\_idiom`.

[32] Horatiu Jula and George Candea. dimmunix: Deadlock immunity system for java/c/c++ software. `http://code.google.com/p/dimmunix/`.

[33] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java concurrency in practice*. Addison-Wesley, 2006.

[34] K. Arun. Semester course project: Deadlock detection in c#. Internal Project Report, 2010.