On deadlocking using Java and C#

Vivek K. Shanbhag[1], Philips Research, PIC, Bangalore.
vivek.shanbhag@philips.com, vivek.shanbhag@gmail.com

September 11, 2007

[1]This enquiry started in early 2005, while the author was at Sun Microsystems with its Java Sustaining team, in the context of problems reported with the JVM version 1.4.2. In the context of the Java Programming Language, and its J2SE API Implementation, a study of the extent of the problem was undertaken and the author was spending 1-day a week on this activity for a period of nearly 6-months. While the study was still incomplete, the author dis-associated from Sun. Now, at Philips Research, the problem is being examined, also, in the context of the C# Programming Language, and its use at PMS.

Contents

Chapter 1

Introduction

This section contains the introductions for both the C# and the Java renderings of this work, so far. It consolidates all the written matter in one place, so that it is kept consistant through its evolution. Eventually, three different writeups shall be dereived from this work:

- A conf. paper: in partial fulfilment of the targets for 2007. This version shall address only the Java version of the study. It shall not divulge any details of the findings from its application to the MIP / PMW Code-bases.
- A Technical Note: in partial fulfilment of the targets for 2007. This version shall be a more complete treatment to the C# version of the study. Including all the details of the study of the MIP & the PMW Code-bases. It could also go to the extent of printing the entire Literate-program that implements that part.
- A Journal Paper: in partial fulfilment of the requirements for the PhD @ IIIT-Bangalore. This work shall additionally address the Java aspect of the problems. This writeup shall Print only the non-MIP and the non-PMW results from the C# study.

This study shall attempt to 'complete' the dynamic aspect of the study. Whereas it may fail short of the intent based upon merely engineering hurdles, that are too uninteresting to handle completely. In which case, we shall proceed with it only a little beyond the point where-at we would have established proof-of-concept, beyond all reasonable doubt.

1.1 Java stuff

1.1.1 Title

A question seeks to be asked: Is a Java-Progammer equipped enough to write a Provably Deadlock-Free (not Race-Free, yet) Java program using the J2SE 1.4.2 API ?

1.1.2 Abstract

The J2SE Java API Enhancements Specification has recently come under the perview of the Java Comunity Process (JCP). Until that happened, a lot of Java-APIs had been designed, and implemented that possibly did not have formal specifications before their implementations were FCSed (First Customer Shipment). The rt.jar package that is available with the J2SE download has continued to grow from ... in J2SE V1.2 to its current ... in the J2SE V1.5 (aka Tiger).

Occassionally some user who runs an otherwise perfectly acceptable multi-threaded program might happen to detect a Java-level deadlock wherein the JVM hangs while trying to execute the byte-code for the corresponding class files. The javadoc pages accessible at the java.sun.com, that specify the Java API Specs do not export information regarding which API methods of public classes are 'synchronised', for instance.

This disables the java-programer from writing a 'provably' deadlock-free program. Moreover, by not commiting itself to this important concurrency aspect of the API-Spec, Sun reserves the right to make any modifications to the iplementation in its future releases. This can, in principle, create the situation wherein a java-program that runs without deadlocking with a given version of the J2SE release, might not continue to do so with some later release.

To correct this situation Sun must do both of the following: (a) Specify the 'synchronised' aspect of the public methods of public classes, and also put a note wherever the implementation of a certain method would need to lock some of its argumernts, and (b) Either be convinced that its API impelemntation doesnot deadlock itself, or publish a list of known deadlocking sceanarios.

In this report we discuss, design and construct a framework that detects certain classes of Java-Level Deadlocks that are reachable for java-programs that use the J2SE API in a completely legitimate manner.

Keywords: Static-checking, Model-Extraction, Java-Reflection API, Literate-Programming.

1.1.3 Introduction

We, in the Java-Sustaining group, every so often, receive user-reported problems wherein some API-abiding java-program demonstrates its ability of causing a Java-Level deadlock. Such problems are often resolved by moving around the 'synchronised' keyword used to 'lock' certain objects when blocks of code are moving it from one consistant state to another, through possibly a sequence of in-consistant states, in the API implementation. Occassionally we also refrain from making that sort of a change to some critical code, citing another workaround for the user to use to work around their problem. We do this to avoid the risk of causing a 'regression' wherein some other unsuspecting user might find his/her code that was working fine with the earlier version of the J2SE, suddenly deadlocking with the newer update release.

In this effort we begin with prototyping, and sketching a framework to detect subclasses of this class of problems. We start with prototyping a program to detect the most simplest subclass of problems in this space. Later we take a counter-example driven approach whereby we grow the programs' ability to detect a larger/additional subclass(es) of problems typified by the counter-example.

In what follows we start by discussing the 'synchronised' aspect of the Java Programing Language, and briefly touch upon its implementation by the JVM. In the next section, we then take an example java-program that is a rather simple multi-threaded program which manages to get into a Java-level deadlock. The subsequent section attempts to use the example program as a basis for discussion about the charachterisation of such programs that can get into deadlocks.

1.1.4 The 'synchronised' keyword of the Java Programming Language

"The Java Language has built-in support for creating multithreaded applications. It uses, [and exposes to the programmer], per-object, and per-class monitor-style locks to synchronise concurrent access to object and class data. [JPL3e]" And, of course, the language-level keyword that allows the programmer to access this ability is 'synchronized'. Chapter 10 of the book discusses various aspects of multi-threading in Java, invluding "Deadlocks" in its section 10.7.

We revisit only a small subset of the subject-matter covered therein, that is directly relevant to the rest of our discussion. "The term *synchronized code* describes any code that is inside a 'synchronized' method or statement". Moreover, the synchronized method could be a 'static' method of the class. That adds up to the following three sceanarios:

- **synchronized methods:** Consider a class 'Account' that is defined below. the methods getBalance, and deposit are implemented so as to invocable upon the same object of type Account, from seperate threads without causing any problems. For instance, when a call to oA.getBalance() is encountered in a thread t1, the execution of t1 will acquire the lock on object oA, (or block for the same, if it must), before executing the java-byte-code corresponding to its method-body. Upon completing the execution of the method-body, the thread t1 will release the lock on object oA.

```
public class Account {
  private double balance;
  public Account (double initialDeposit)        { balance = initialDeposit; }
  public synchronized double getBalance ()      { return balance; }
  public synchronized void deposit (double amnt) { balance += amnt; }
}
```

- **static synchronized methods:** Static methods of a class could also be declared synchronized. When a thread encounters a call to such a method, its execution acquires a lock on the Class object corresponding to the containing class, (or blocks for the same, if it must), before executing the java-byte-code corresponding to its method-body. Upon completing the execution of the method-body, the thread releases the lock on the Class object.

- **synchronized statements:** "The synchronized statement enables one to execute synchronized code that acquires the lock of any object, not just the current object, or for durations less than the entire invocation of a method. It has two parts: an object whost lock is to be acquired and a statement to execute when the lock is obtained. Its general form is given below, wherein the *expr* must evaluate to an object reference. When the lock is obtained, the *statements* in the block are executed. At the end of the block the lock is released – if an uncaught exception occurs within the block, the lock is still released.

  ```
  synchronized (expr) { statements }
  ```

1.1.5 A Java program that locks the 1.4.2_04 JVM

Consider the following java program:

"DeadlockTest.java" 1 ≡

```
import java.io.IOException;

public class DeadlockTest    {
    class T1 extends Thread  {
        Properties sp;
        public T1 (java.util.Properties p)  {  sp = p;                         }
        public void run()    {
            try                 {  sp.store(System.out, null);              }
            catch (IOException e) {  System.out.println("IOException : " + e);  }
        }
    }
    class T2 extends Thread  {
        public void run()        {  java.util.TimeZone.getTimeZone("PST");     }
    }
    public static void main(String[] args)  {  new DeadlockTest();              }
    public DeadlockTest()    {
        new T1(System.getProperties()).start(); //       t1.start();
        new T2().start();                       //       t2.start();
    }
}
◇
```

On Compiling & executing the above program (javac DeadlockTest.java; java DeadlockTest), it promptly deadlocks the JVM, which can be then interrupted by sending it the ctrl+"̈signal. On doing so the following infromation is printed by the JVM before it continues to remain deadlocked.

⟨ Thread dump from the deadlocked JVM 2 ⟩ ≡

```
Full thread dump Java HotSpot(TM) Client VM (1.4.2_04-b05 mixed mode):

"DestroyJavaVM" prio=1 tid=0x08052098 nid=0x3d6f waiting on condition [0..bfffc924]
```

```
"Thread-1" prio=1 tid=0x080a3fe8 nid=0x3d6f waiting for monitor entry [4cf05000..4cf06298]
        at java.util.Hashtable.get(Hashtable.java:332)
        - waiting to lock <0x44740560> (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 <0x48849078> (a java.lang.Class)
        at sun.util.calendar.ZoneInfo.getTimeZone(ZoneInfo.java:524)
        at java.util.TimeZone.getTimeZone(TimeZone.java:448)
        at java.util.TimeZone.getTimeZone(TimeZone.java:444)
        - locked <0x48844c50> (a java.lang.Class)
        at DeadlockTest$Thread2.run(DeadlockTest.java:36)

"Thread-0" prio=1 tid=0x080a47e8 nid=0x3d6f waiting for monitor entry [4ce86000..4ce86318]
        at java.util.TimeZone.getDefault(TimeZone.java:498)
        - waiting to lock <0x48844c50> (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 <0x44740560> (a java.util.Properties)
        at DeadlockTest$Thread1.run(DeadlockTest.java:27)

"Signal Dispatcher" daemon prio=1 tid=0x0809cd00 nid=0x3d6f waiting on condition [0..0]

"Finalizer" daemon prio=1 tid=0x08088480 nid=0x3d6f in Object.wait() [4c921000..4c921598]
        at java.lang.Object.wait(Native Method)
        - waiting on <0x44740490> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:111)
        - locked <0x44740490> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:127)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=1 tid=0x08087890 nid=0x3d6f in Object.wait() [4c8a1000..4c8a1618]
        at java.lang.Object.wait(Native Method)
        - waiting on <0x44740380> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:429)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:115)
        - locked <0x44740380> (a java.lang.ref.Reference$Lock)

"VM Thread" prio=1 tid=0x08086570 nid=0x3d6f runnable

"VM Periodic Task Thread" prio=1 tid=0x0809f598 nid=0x3d6f waiting on condition
"Suspend Checker Thread" prio=1 tid=0x0809c360 nid=0x3d6f runnable

Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x080890fc (object 0x44740560, a java.util.Properties),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x08089134 (object 0x48844c50, a java.lang.Class),
  which is held by "Thread-1"
```

```
Java stack information for the threads listed above:
===================================================
"Thread-1":
        at java.util.Hashtable.get(Hashtable.java:332)
        - waiting to lock <0x44740560> (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 <0x48849078> (a java.lang.Class)
        at sun.util.calendar.ZoneInfo.getTimeZone(ZoneInfo.java:524)
        at java.util.TimeZone.getTimeZone(TimeZone.java:448)
        at java.util.TimeZone.getTimeZone(TimeZone.java:444)
        - locked <0x48844c50> (a java.lang.Class)
        at DeadlockTest$Thread2.run(DeadlockTest.java:36)
"Thread-0":
        at java.util.TimeZone.getDefault(TimeZone.java:498)
        - waiting to lock <0x48844c50> (a java.lang.Class)
        at java.text.SimpleDateFormat.initialize(SimpleDateFormat.java:500)
        at java.text.SimpleDateFormat.<init>(SimpleDateFormat.java:443)
        at java.util.Date.toString(Date.java:981)
        at java.util.Properties.store(Properties.java:531)
        - locked <0x44740560> (a java.util.Properties)
        at DeadlockTest$Thread1.run(DeadlockTest.java:27)

Found 1 deadlock.
```
◇

Macro never referenced.

One can observe from the "Java stack information for the threads" listed (immediately) above that the first thread ("Thread-1") has called a static synchronised method "java.util.TimeZone.getTimeZone" which calls a stack of functions, (including another static synchronised method "sun.util.calendar.ZoneInfoFile.getZoneInfo") which calls finally a synchronised function "java.util.Properties.getProperty" on a particular Properties Object that has been locked by the second Thread ("Thread-0"). This other thread has begun by firstly locking the same particular Properties object (due to its call to the synchronized method "java.util.Properties.store") and then gone on to call a stack of functions that eventually calls the static synchronised method "java.util.TimeZone.getDefaul which requires to lock the very same "java.lang.Class" Object that was firstly locked by the first thread. Thus causing the Java-level Deadlock.

1.1.6 Discuss

Section 10.7 of [JPL3e] discusses such "Deadlocks" with an example, and further, observes: "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. ...*"

Trying to understand the above quote in the context of our discussion so far raises interesting questions. For instance,

- which programmer should attempt to avert the program in question from reaching its deadlock. Is it the application progremmer, or is it the programmer(s) that provide the J2SE API Implementation for the application programmer to use. Or does it include both.
- (Assuming any of the three possibilities proposed above, would still boil down to java-programmers (be they inside/outside SUN Microsystems)). Does the *resource ordering* technique proposed in the book

apply to the problem at hand. Is there enough information publicly available that makes it possible to readily apply those techniques in the context of the class of problems being discussed here. Is it realistic to do that, given that there is a new update of J2SE that is FCSed approximately once every threee months. Does such a methodology scale for large applications.

- Are there other alternatives. What other methods / tools / techniques exist that can be put to use in the context being discussed.

The example program above was originally raised as an Customer-Escallation by one of the Source-licensees of the J2SE Source-code, and was fixed in its 1.4.2_07 release. The above test-case now *fails* to deadlock, for instance, the 1.4.2_07 release of the JVM, using the associated rt.jar.

For completeness we merely mention the source-code changes made to the J2SE API Implementation that "fixes" the problem. This fix was not deviced using any formal analysis of the J2SE Source-base. It was deviced (by an engineer knowledgable about the design criterion underlying the implementation of the classes: java.util.TimeZone, and java.util.Properties), by looking at (among other things) the stacks of the two threads discussed above, as they appear in the output listed above.

⟨ Source-diffs 3 ⟩ ≡

```
/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();
      }
```
◇

Macro never referenced.

So as is apparent from the above, in this instance, the J2SE Implementation did accept responsibility for the DeadlockTest program's ability to reach a Java-level deadlock state, and took whatever corrective action it deemed fit. The Customer was able to accept the fact that the next release of the JDK-1.4.2 would fix their problem. However, this need not happen similarly always. In a subsequent section we attempt to address some of the questions raised above.

1.1.7 Sample Java, and the disassembly of its corresponding class-file

In this section we shall briefly look at a short Java-class and the disassembly of its corresponding class-file. We use 'javap' a tool provided by Sun-Microsystems as a part of its standard J2SE distribution, for both, compiling Java, and disassembly of the so generated class-file, containing byte-code.

$\langle$ A Short Java Program 4 $\rangle \equiv$

```
import java.io;
package mlTest;
class MLtest  {
  class NmlTest {
    static synchronized void main (string[] args) {
      System.out.println ("The Max Integer value: " + Integer.MAX_VALUE);
    }
  }
}◇
```

Macro never referenced.

After writing the above contents into a "MLtest.java" file, the command "javac MLtest.java" compies the file into a class file called "MLtest.class". The command "javap -c -classpath . MLtest" generates the disassembled version of the information in the class file which is in plain-text. Below we reproduce the same so as to give the reader a feel for the format of the text that we might be able to parse to do our processing in the rest of this write-up.

$\langle$ The disassembled version of MLtest.class 5 $\rangle \equiv$

```
Compiled from "MLtest.java"
public class mlTest.MLtest extends java.lang.Object{
public mlTest.MLtest();
  Code:
   0: aload_0
   1: invokespecial  #1; //Method java/lang/Object."<init>":()V
   4: return

public static synchronized void main(java.lang.String[]);
  Code:
   0: getstatic   #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3: ldc   #3; //String The Max Integer value: 2147483647
   5: invokevirtual  #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8: return

}◇
```

Macro never referenced.

1.2 The C# work

1.2.1 Title

On the deadlocking possibilities in MIP (C# Source)

1.2.2 Abstract

Commonly used *Unmanaged Languages* like C & C++ did not make it naturally easy to do Multi-threading while designing / programming applications. It was also a problem, then, that most well-understood & commonly-used API-implementations (Library-code) were not MT-safe, nor multi-threaded themselves. *Managed Languages* like Java, and C# have changed that by incorporating threading into the Standard API, and providing synchronisation primitives in the language to facilitate its correct use.

However, there is evidence of perfectly legitimate Java-Programs, for instance, that are conformant to both, the language-spec, and the J2SE-API-Spec, deadlock-ing the hosting JVM-instance. This reveals that both these languages, and their programming frameworks could benefit from fresh thought into the design of the language-feature, *or* of its systamatic & correct use in implementing library-code. Since we, at Philips (PMS), use both C# & Java, to implelemt our MIP-code-base, for instance, that is a component in our Modality-implementations, it will be only wise for us to enquire if our use of these managed languages does pottentially expose us to these sort of problems. And if so, what are our options?

Our group, at Philips Research, is trying to address the class of problems that software-development teams can be faced with in their attempts to design & build software that is comprising of Commercial Off-The-Shelf (COTS) Components. From prior experience, and literature, we are aware of occassions wherein the JVM (Java Virtual Machine) is known to have deadlocked for no fault of the Application-programmer. Such occurrances often result from the interference of two features: Multi-Threading, and Synchronisation. Both these features are present in C#, and more-or-less similarly.

MIP (Medical Imaging Platform) is one such (Philips-owned) COTS-Component that is pottentially reusable accross all the PMGs (Philips Modality Groups) inside of PMS (Philips Medical Systems). MIP is implemented mostly using C# & the associated .NET framework. This currently undertaken brief exercise attempts to inquire if the it does indeed use both these features, thereby creating the pottential deadlocking possibility that Modalities integrating with it would, therefore, get exposed to.

Keywords: Static-checking, MSIL-Disassembly, Reflection API, Literate-Programming.

1.2.3 Introduction

In what follows we start by discussing the 'lock' aspect of the C# Programing Language, and briefly touch upon its apperant implementation by the CLR. We then specify the user-interface to the utility. The next section explores the design space, some. Having thus understood our task at hand, we then proceed to develop the implementation, in parts, and assemble them to fetch the desired tool, in the second chapter.

1.2.4 The 'lock' keyword of C#

The 'lock' keyword of the C# Language is used to protect the state of a shared object from being cloberred due to concurrent acces from multiple threads of execution. It is used as follows:

lock (*expr*) *statement*

The above program-fragmant is compiled into MSIL-code, which, when disassembled fetches the following structure:

⟨ Disassembled MSIL corresponding to the above 6 ⟩ ≡

```
... code evaluating the ‘expr’ part.
IL_000b:  ldloc.0
IL_000c:  call void class [mscorlib]System.Threading.Monitor::Enter(object)
.try { // 0
  ... code corresponding to the ‘statement’ part.
} // end .try 0
finally  { // 0
  IL_0036:  ldloc.0
  IL_0037:  call void class [mscorlib]System.Threading.Monitor::Exit(object)
  IL_003c:  endfinally
} // end handler 0
◇
```

Macro never referenced.

1.2.5 Sample C#, and the disassembly of its corresponding MSIL

Some of the information the program seeks to discover can be fetched elegantly using the Reflection API. For instance, the list of all the 'Classes' contained in a given Assembly, and the list of all the 'Methods' associated with a given 'Class'. However, the abilities of the Reflection API end there (as far as our needs are concerned). It cannot help us to identify those methods that do use the 'lock' construct, in its implementation. Nor can it help us in identifying those methods that do invoke 'theading' calls, and which.

The complete information that we desire can be discovered by inspecting the "disassembled assembly-modules". The compiler for the C# Language, called *mcs* on GNU/Linux systems generate '.dll' or '.exe' files that contain the MSIL-code corresponding to collections of Class-files. Another tool, from the 'mono-Project' collection, called *monodis* parses MSIL assemblies to generate a textual rendering of it, that is rather easily parsed. The most advantageous part of using this 'apparently convoluted' route of first compiling, and then disassembling the code to parse and discover the required information is that we need to write **no error-handling code**. We can be reasonably certain that the C#-Compilers would have already trapped any errors in the corresponding source, and only after the source is rid of them does our tool get into the picture.

Before we embark onto development of our program we briefly look at a short C# program, and the corresponding textual MSIL Information.

⟨ A Short C# Program 7 ⟩ ≡
```
using System;
namespace MLtest {
  class MLtest   {
    class NmlTest {
      [STAThread] static void Main (string[] args) {
        lock (typeof(MLtest.NmlTest)) {
          System.Console.WriteLine ("The Max 64-bit Integer: " +
                                    Int64.MaxValue.ToString());
        }
      }
    }
  }
}◇
```
Macro never referenced.

Upon compiling the above C# code, using mcs of the mono-toolkit on the GNU/Linux platform, and then disassembly-ing the so-generated MSIL code, one fetches the following:

⟨ Corresponding Disassembled MSIL 8 ⟩ ≡
```
.assembly extern mscorlib
{
  .ver 1:0:5000:0
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 ) // .z\V.4..
}
.assembly 'MLtest'
{
  .hash algorithm 0x00008004
  .ver  0:0:0:0
}
.module MLtest.exe // GUID = {71806EE1-17CB-4A58-BD56-566DC63FE22F}


.namespace MLtest
{
  .class private auto ansi beforefieldinit MLtest
   extends [mscorlib]System.Object
  {
    // method line 1
    .method public hidebysig  specialname  rtspecialname
           instance default void .ctor ()  cil managed
    {
        // Method begins at RVA 0x20ec
   // Code size 7 (0x7)
   .maxstack 8
   IL_0000:  ldarg.0
   IL_0001:  call instance void object::.ctor()
   IL_0006:  ret
    } // end of method MLtest::instance default void .ctor ()

    // method line 2
    .method private static  hidebysig
           default void Main (string[] args)  cil managed
    {
        .custom instance void class [mscorlib]System.STAThreadAttribute::.ctor() =  (01 00 00 00 ) // ....

        // Method begins at RVA 0x20f4
   .entrypoint
   // Code size 62 (0x3e)
   .maxstack 8
   .locals init (
      class [mscorlib]System.Type   V_0,
      int64 V_1)
   IL_0000:  ldtoken MLtest.MLtest
   IL_0005:  call class [mscorlib]System.Type class [mscorlib]System.Type::GetTypeFromHandle(valuetype [mscorl
   IL_000a:  stloc.0
   IL_000b:  ldloc.0
   IL_000c:  call void class [mscorlib]System.Threading.Monitor::Enter(object)
   .try { // 0
     IL_0011:  ldstr "The Max 64-bit Integer: "
     IL_0016:  ldc.i8 0x7fffffffffffffff
     IL_001f:  stloc.1
     IL_0020:  ldloca.s 1
     IL_0022:  call instance string int64::ToString()
     IL_0027:  call string string::Concat(string, string)
     IL_002c:  call void class [mscorlib]System.Console::WriteLine(string)
     IL_0031:  leave IL_003d

   } // end .try 0
   finally  { // 0
     IL_0036:  ldloc.0
     IL_0037:  call void class [mscorlib]System.Threading.Monitor::Exit(object)
     IL_003c:  endfinally
   } // end handler 0
   IL_003d:  ret
    } // end of method MLtest::default void Main (string[] args)

  } // end of class MLtest.MLtest
}◇
```

Macro never referenced.

In the above text, notice that the fully-qualified names of all called methods appear in a predictable & detectable contextual form. Similarly, the use of the 'lock' construct is translated as described in section 2. Below we then proceed to think through the possible implementation options, and develop it along the way, as we go.

Chapter 2

Investigating the Problem-Size

In this part of our work, we develop short programs that help us investigate C# and Java Language Libraries with a view to find the number of concrete instances that possibly could contribute to the problem of deadlocks. In the programs we develop below, we investigate libraries to find their usage of specific language features that we are interested in.

Below, we begin with a section that specifies the 'symptoms' that we will investigate for. Then in the next section we develop a Java program to investigate the problems with C# Language libraries. The next section develops a similar program for the Java Language Libraries. Finally we present the results of running these programs on typically used Language Libraries for both C#, and Java.

2.1 Charachterising code-base that *could* deadlock ?

The answer to the above may not be obviously stated in a closed-form manner. So let us explore the inverse questoin, to begin with: What could be the charachterisics of a code-base that can not possibly deadlock? There are at-least three classes of software that can be listed here, readily:

1. A C#/Java Application that doesnot start any threads at all, is clearly a good example of this category. Such applications execute in a single thread: that which started with a call to the "main" function of the target class. If one explores the transitive closure of all the classes referenced by this target-class, through its various members, and through the types of the local-variables of its methods, and the methods it invokes, from its main method, ... Now, if it turns out that none of the code associated with all the types from this closure-set refers to any threading calls, not are any of the classes therein inheriting from the System.Threading.Thread class, then we are done establishing what we set out to do.

2. A C#/Java application that invokes only such code that does not use the *lock()* construct from C# (or the *synchronised* construct from Java) in any of the methods associated with the types within the 'transitive closure', as computed from the above, is another good example (at the other extreme of the spectrum, though) of code that cannot possibly deadlock. Notice that in this case, we place no restrictions on the use of threading in the application. But the threads simply donot lock anything, and consequently do not await any resource, thereby never deadlocking.

3. To now look at a slightly more interesting (non-trivial) class of mlti-threaded applications, let us consider an application that uses both the multi-threading and the sychronisation features of the language. However, may it have an interesting property whereby if one were to traverse the call-graph, starting from the 'main' method of the target-class within the code in our transitive-closure set, one finds that, along any path that leads from the starting point to a leaf-method (where the path ends), one encounters at-most one use of the lock/synchronised language-feature. Clearly, such code, can not deadlock either, since in any of its threads, if once a lock is acquired, that thread doesnot await any other resource, until it releases the one it has acquired.

4. An even more interesting example of such non-deadlocking application is this: Consider an application which has been programmed as follows: it uses multi-threading; it uses synchronisation; however, it also insists that in any of its threads, once a lock has been acquired, (and until it is released), only such other objects can be synchronised upon, which were created after the most-recently-synchronised-upon lock. Clearly, this property of an application is not detectable statically (unlike the ones above). But, one could, in principle, either write a monitor, or a minor modification to the class-heirarchy, or modify the virtual-machine (in the worst-case) to firstly tag every new allocation (object-creation) with the timestamp when it was created, and then also ensure that any lock acquisition by a thread that already has locks ensures the required *Ordering-property.* Clearly, such an application cannot deadlock ?

Notice that the above discussion only discussed the case of deadlocks with *Applications*, since they all started with the target-class that provided the main-method. Whereas in the case of library code-base, there is no main-function. However, there could be library-implementations such that they manage to (pottentially) deadlock whatever application they are *imported* into. And, in the context of libraries, this is precisely what we would like to establish: Given a library-implementation, has it been done so as to be inherently problematic to any application that might choose to use it ? We defer trying to answer this question, yet. However, observe that if we have a bunch of library-code, and it is known to be refering to multi-threading, (starting, controlling threads), and synchronisation (using lock()/synchronized constructs), then we would be justified in at-least viewing the use of such a library with some amount of apprehension, w.r.t deadlocking. Below we list the three criteria that we use to inquire as to whether a library code-base is "suspect" to pottentially contribute towards an deadlocks in applications that choose to use them:

1. The use of any functions from the *System.Threading* namespace in the *implelemtation* of the methods (the method body) contained in the classes from the library.
2. The use of *lock()* in the implementation of any methods, like above.
3. The presence of any classes that inherit from the *System.Threading.Thread* class in the library. Notice that this last property, and the list of classes that honour it, can be detected using the reflection API, unlike in the case of the previous two properties in this list.

Merely from an implementation point of view, there is another distinction that would be interesting to notice between the C# and the Java cases: Nested uses of the *lock()* construct within a method-implementation is easy to detect, and catagorise precisely in C#, however in Java, the corresponding *synchronized()* construct translates into a pair of *monitorEnter* and *monitorExit* statements, and their enclosed code does not necessarily follow the single exit criterion. That makes it slightly difficult to demarkate precisely the code-peices that are enclosed by nested such uses in corresponding source. The item 2 from the above list therefore is split into the list-items below in the Java-Language case:

1. Methods that are tagged by the *synchronized* tag.
2. *static* methods that are tagged by the *synchronized* tag.
3. Methods that use a single instance of the *synchronized()* construct, that gets translated into a single pair of *monitor -Enter & -Exit* byte-codes in the implementation of the methods, in the corresponding class-file.
4. Methods from the class-files that use a single instance of the *monitorEnter* byte-code but with more than one occurrance of the *monitorExit* byte-code.
5. Methods from the class-files that have multiple instances of *monitorEnter* byte-code.

2.2 Implementation: C# case

In this section, we firstly discuss the design of the user-interface of a tool that would help us to guage the size of the problem given a specific library. Having done that we then develop the implementation in three parts as follows: in one section we look at the code to fork out a disassembler process and read its output; in another we develop the code to handle the statistical reporting aspect of the requirement; and then we detail the code to parse the text, and populate the variables that get reported. Then, lastly, we put together the code to do the job for us.

2.2.1 Usage

We begin by specifying the usage specification for the desired program and its (program development) environment, to some extent. The program usage should be:

java lockNDthread [-a|-n|-c|-m] dll-or-exe-name [dll-or-exe-name-list]

The name of the program is (say) 'lockNDthread'. It must be invoked with at-least one argument: the path of a C# Assembly, dll or exe. It shall report all uses of threading & locks in the MSIL code that composes the assembly. When it is invoked with more fully-specified Assembly path-names, it shall report the same for all of them. The options mean:

- **-a**: (default) report usage, consolidated at Assembly-level.
- **-n**: report usage, collected at Namespace-level.
- **-c**: report usage, collected at Class-level.
- **-m**: report usage, detailed at Method-level.

C# is a language that is (currently) best-supported on the Microsoft platforms. We tried using the Open-Source 'mono & mcs' tool-kit on the RedHat GNU/Linux system, but we could not get our program to run on large input-data sizes. Reading large input files into the program seemed to triger an out-of-memory state in the mono virtual machine. We shall therefore choose to use java on GNU/Linux, to analyse our C# assemblies. Additionally, we shall use the Literate Programming approach to design & develop the program.

In the sample usage below, we invoke the default mode of the utility, corresponding to the Assembly-Level consolidation. The output reports on findings per assembly. Only those assemblies are reported upon which test lock/thread possitive. The final total-line reports the total number of methods, classes, and Namespaces visited, and the portion of the input assembiles that were found to be using the feature.

⟨ Sample Usage & Output 9 ⟩ ≡
```
vivek> java lockNDthread as1.dll as2.dll as3.dll application.exe
as1.dll:
    4 lock() calls, 16 'Thread.*' calls in 1091 Methods in 121 Classes in 85 Namespaces.
as2.dll:
    0 lock() calls, 2 'Thread.*' calls in 146 Methods in 20 Classes in 20 Namespaces.
application.exe:
    0 lock() calls, 985 'Thread.*' calls in 14083 Methods in 1371 Classes in 1260 Namespaces.
Total:
    4 lock() calls, 903 'Thread.*' calls in 15944 Methods, 1933 classes, in 1365 Namespaces, in 3/4 Assemblies
◇
```
Macro defined by 9, 10, 11, 12.
Macro never referenced.

In another sample usage below, we invoke the namespace-consolidation mode of the utility. The output reports on findings per Namespace; only those Namespaces are reported upon (also naming the assembly that contained it) which test lock/thread possitive. The final total-line reports the total number of methods, classes visited, and the portion of the Namespaces found to have been using the feature.

⟨ Sample Usage & Output 10 ⟩ ≡

```
vivek> java lockNDthread -n as1.dll as2.dll
Namespace1 (as1.dll):
    3 lock() calls, 12 ‘Thread.*’ calls in 41 Methods in 3 Classes.
Namespace3 (as1.dll):
    1 lock() calls, 4 ‘Thread.*’ calls in 21 Methods in 2 Classes.
Namespace44 (as2.dll):
    0 lock() calls, 2 ‘Thread.*’ calls in 46 Methods in 4 Classes.
Total:
    4 lock() calls, 18 ‘Thread.*’ calls in 108 Methods, 9 classes, in 3/105 Namespaces.
◇
```

Macro defined by 9, 10, 11, 12.
Macro never referenced.

In yet another sample usage, below, that invokes the utility in its Class-level-Consolidated reporting mode, we fetch output that is similar to the one above, an the next (lower) level of abstraction.

⟨ Sample Usage & Output 11 ⟩ ≡

```
vivek> java lockNDthread -c as1.dll as2.dll
Class1 (Namespace1 (as1.dll)):
    1 lock() calls, 6 ‘Thread.*’ calls in 11 Methods.
Class2 (Namespace1 (as1.dll)):
    1 lock() calls, 2 ‘Thread.*’ calls in 9 Methods.
Class3 (Namespace1 (as1.dll)):
    1 lock() calls, 4 ‘Thread.*’ calls in 9 Methods.
Class59 (Namespace3 (as1.dll)):
    1 lock() calls, 2 ‘Thread.*’ calls in 7 Methods.
Class77 (Namespace3 (as1.dll)):
    0 lock() calls, 2 ‘Thread.*’ calls in 7 Methods.
Class102 (Namespace44 (as2.dll)):
    0 lock() calls, 2 ‘Thread.*’ calls in 6 Methods.
Total:
    4 lock() calls, 18 ‘Thread.*’ calls in 49 Methods, 6/9 classes.
◇
```

Macro defined by 9, 10, 11, 12.
Macro never referenced.

⟨ Sample Usage & Output 12 ⟩ ≡

```
vivek> java lockNDthread -m as2.dll
???
◇
```

Macro defined by 9, 10, 11, 12.
Macro never referenced.

2.2.2 Disassemble Binaries into Text

In our program (fragment below) through forking an external process that invokes the *monodis* command with the input parameters (list of assemblies) we enable the processing code in the main thread. The main thread, then, opens the output stream of the child process for reading in the text-rendering of the MSIL from the assemblies.

⟨ Fork Disassembler, prepare to read text-MSIL 13 ⟩ ≡

```
static public BufferedReader
populateTextReader (String[] params)  {
    try {
        String arg = "";
        for (String s : params)
            if (s.charAt(0) != '-')  arg += " " + s;
        Process child = Runtime.getRuntime().exec ("/usr/local/bin/monodis " + arg);
        InputStream disOut = child.getInputStream();
```

```
            InputStreamReader r = new InputStreamReader (disOut);
            return new BufferedReader (r);
            // while ((line = textIn.readLine()) != null)
        } catch (Exception e) {
            System.out.println (e.toString());
        }
        return null;
    };◇
```

Macro referenced in 47.

Notice that we have arranged the above method as a public member of the class that we will develop. We intend that the above method would be called within the main-method of the class.

2.2.3 Data Aggregation & Reporting

We firstly write the following peice of code to read the invocation options to find out what level of consolidation is requested in the collection & reporting of statistics. We store this information in a static variable of type 'char' in the class.

⟨ Read Consolidation Level 14 ⟩ ≡

```
    if (args[0].charAt(0) == '-')  {
        conLevelAt = args[0].charAt(1);
        if (conLevelAt != 'm'  &&  conLevelAt != 'c'  &&
            conLevelAt != 'n'  &&  conLevelAt != 'a')  {
            System.out.println ("Usage: java LockNdThread -[mcna] assembly-name-list");
            return;
        }
    }
    ◇
```

Macro referenced in 47.

Below we define the 'state' that collects the information to be reported at the end of the run. We populate it incrementally as we go along. Also while we statically instantiate all the variables, we only populate those that need to be reported upon, based upon the consolidation-Level requested.

- **ltPconLvlEntities**: This integer variable counts the number of entities that are detected to be *lock / thread Positive*. The choice of entities is made by the user through choosing the cut-off level, at which the consolidation is requested. For instance, the use of the '-c' invocation flag would cause the number of *infected* classes in the assemblies visited.

- **assemblies**: This integer counts the total number of .net-assemblies seen in the input. Similarly, the counters **cumNspaces, cumClasses, cumMethods** count the total number of NameSpaces, Classes and Methods visited, respectively. The variables **cumLocks, & cumUses** count the number of uses of the *lock()* and any *Thread.** function-call, respectively. The values of some appropriate subset of the counters discussed in this point, and the previous one are reported on the last line of the output of this tool, as is seen in the sample usages discussed earlier.

- **aNspaces, aClasses, aMethods, aLocks, aUses**: These counters are similar to the ones discussed in the previous point, except that they count the occurrances of the corresponding entities, respectively, at a per-assembly level. They are updated and reported on a per-assembly basis (and reset), for those runs where the '-a' (default) invocation flag is used.

- **nsClasses, nsMethods, nsLocks, nsUses**: These counters are similar to the ones discussed above, except that they count the occurrances of the corresponding entities, respectively, at a per-NameSpace level. They are updated and reported on a per-NameSpace basis (and reset), for those runs where the '-n' invocation flag is used.

- **cMethods, cLocks, cUses**: These counters count the occurrances of their corresponding entities, respectively, at a per-Class level. They are updated and reported on a per-Class basis (and reset), for those runs where the '-c' invocation flag is used. For the (only other) invocation flag "-m", the associated variables are **mLocks, & mUses**.

- **lopens**: This is a counter that is used to remember the number of times we have seen the begining of the code-idiom that corresponds to the programmers use of the *lock()* construct. Recollect that we need to see also the corresponding closing-part of the code-idiom to be able to up the counter 'mLocks'. We therefore decrement 'lopens' everytime we are able to increment 'mLocks'. If we get to the end of the method-body, with a non-zero count held in 'lopens', we accumulate that value into the 'mUses' counter.

⟨ Statistics Collecting State 15 ⟩ ≡

```
static int ltPconLvlEntities = 0, assemblies = 0;
static int cumNspaces = 0, aNspaces = 0;
static int cumClasses = 0, aClasses = 0, nsClasses = 0;
static int cumMethods = 0, aMethods = 0, nsMethods = 0, cMethods = 0;
static int cumUses = 0, aUses = 0, nsUses = 0, cUses = 0, mUses = 0;
static int cumLocks = 0, aLocks = 0, nsLocks = 0, cLocks = 0, mLocks = 0, lopens = 0;
◇
```

Macro referenced in 47.

We update the state described above at the close of every method as below. In case Method-level reporting is requested, we report, iff there is non-zero values to report. Notice that the counters reported on the "Total:" line are updated unconditionally.

⟨ Statistics at Method-close 16 ⟩ ≡

```
mUses += lopens;  lopens = 0;
if (conLevelAt == 'm'  &&  (mLocks>0 || mUses>0))  {
    ltPconLvlEntities++;
    System.out.println (methodName + "." + className + "(" + nameSpace + "(" + fileName + ")):");
    System.out.println ("    " + mLocks + " lock() calls, " + mUses + " 'Thread.*' calls.");
}  else if (conLevelAt == 'c'  ||  conLevelAt == 'n'  ||  conLevelAt == 'a')  {
    cUses += mUses;  cLocks += mLocks;  cMethods++;
}
cumUses += mUses;  cumLocks += mLocks;  cumMethods++;  mUses = 0;  mLocks = 0;
◇
```

Macro referenced in 42.

At Class-close events we do the following, similar to the above. And further, at NameSpace-close, and Assembly-close, we update state similarly.

⟨ Statistics at Class-close 17 ⟩ ≡

```
if (conLevelAt == 'c'  &&  (cLocks>0 || cUses>0))  {
    ltPconLvlEntities++;
    System.out.println (className + "(" + nameSpace + "(" + fileName + ")):");
    System.out.println ("    " + cLocks + " lock() calls, " + cUses +
        " 'Thread.*' calls in " + cMethods + " Methods.");
}  else if (conLevelAt == 'n'  ||  conLevelAt == 'a')  {
    nsUses += cUses;  nsLocks += cLocks;  nsMethods += cMethods;  nsClasses++;
}
cumClasses++;  cUses = 0;  cLocks = 0;  cMethods = 0;
◇
```

Macro referenced in 38.

⟨ Statistics at Namespace-close 18 ⟩ ≡

```
if (conLevelAt == 'n'  &&  (nsLocks>0 || nsUses>0))  {
    ltPconLvlEntities++;
    System.out.println (nameSpace + "(" + fileName + "):");
    System.out.println ("    " + nsLocks + " lock() calls, " + nsUses +
        " `Thread.*' calls in " + nsMethods + " Methods in " + nsClasses + " Classes.");
}  else if (conLevelAt == 'a')  {
    aUses  += nsUses;       aLocks += nsLocks;
    aMethods += nsMethods;  aClasses += nsClasses;  aNspaces++;
}
cumNspaces++;  nsUses = 0;  nsLocks = 0;  nsMethods = 0;  nsClasses = 0;
◇
```

Macro referenced in 34.

⟨ Statistics at Assembly-close 19 ⟩ ≡

```
if (conLevelAt == 'a'  &&  (aLocks>0 || aUses>0))  {
    ltPconLvlEntities++;
    System.out.println (fileName + ":");
    System.out.println ("    " + aLocks + " lock() calls, " + aUses +
        " `Thread.*' calls in " + aMethods + " Methods in " + aClasses +
        " Classes in " + aNspaces + " Namespaces.");
}
assemblies++;  aUses = 0;  aLocks = 0;  aMethods = 0;  aClasses = 0;  aNspaces = 0;
◇
```

Macro referenced in 33.

Finally we report the "Total:" line counters, as below.

⟨ Cumulative Statistics at Main-close 20 ⟩ ≡

```
System.out.println ("Total:");
System.out.print ("    " + cumLocks + " lock() calls, " + cumUses + " `Thread.*' calls in ");
if (conLevelAt == 'm')
    System.out.println (ltPconLvlEntities + "/" + cumMethods + " Methods.");
else {
    System.out.print   (cumMethods + " Methods, in ");
    if (conLevelAt == 'c')
        System.out.println (ltPconLvlEntities + "/" + cumClasses + " Classes.");
    else {
        System.out.print   (cumClasses + " Classes, in ");
        if (conLevelAt == 'n')
            System.out.println (ltPconLvlEntities + "/" + cumNspaces + " NameSpaces.");
        else  {
            System.out.print   (cumNspaces + " NameSpaces, in ");
            System.out.println (ltPconLvlEntities + "/" + assemblies + " Assemblies.");
        }
    }
}◇
```

Macro referenced in 47.

2.2.4 The control structure of the Program

In this section, we develop the aspect of the code that parses the input data streem to identify the information of interest, and points in the streem where we would update our statistics appropriately.

1. A quick look at the disassembled output reveals a simple structure. Below we discuss it, alonside developing code to extract the required information. To begin with, we expect to receive the textual MSIL information through a member variable `textIn`, of type `BufferedReader` (populated by the function `populateTextReader()`, developed in *Fork Disassembler, prepare to read text-MSIL*).

⟨ Begin processing input Text 21 ⟩ ≡

```
    String line = textIn.readLine();◇
```

Macro referenced in 47.

2. **.assembly extern**: declarations at the begining, 5-lines per 'import'ed namespace. This information is not of any consequence, so we write no code for it.

3. **.assembly 'name'**: next, 5-lines. This is read and the name retained, after ignoring any number of lines preceeding it. Every occurrance of this declaration is followed by one (or more) 'namespace' declaration(s) discussed below. We develop a function that is able to read the entire assembly definition.

   The end of this definition is flagged bt the start of yet another assembly-definitions just like this one. So in the below functionality we read past the end of the assembly, into the first line of the next (or null, if this was indeed the last one, during this execution) and return the last line read back to the caller, so that it can be processed by the next call to this functionality.

⟨ Collect Entire Assembly 22 ⟩ ≡

```
    static String readAssembly (String line)
        throws IOException {
        ⟨ Extract (23 ".assembly '",24 " ",25 assemblyName,26 null ) 27 ⟩
    ◇
```

Macro defined by 22, 28, 33.
Macro referenced in 47.

We define a convinience code-fragment that extracts the required information (demarcated by the character identified by the second parameter) from a line (matching the prefix supplied as the first parameter) into an identified variable (named by the the third parameter). In case of any error in the input, the value specified as parameter 4, if any, is returned.

⟨ Extract 27 ⟩ ≡

```
    while (line!=null && line.indexOf(@1)<0)
        line = textIn.readLine();
    if (line==null || ((t = line.split(@2)).length < 2)) {
        System.out.println ("Input '@3' Missing!");
        return @4;
    }
    @3 = t[t.length - 1];
    // System.out.println("Read @3: " + @3);
    ◇
```

Macro referenced in 22, 28, 34, 38, 42.

4. **.module name.exe // ...**: next. This line is read and 'name.exe' retained.

⟨ Collect Entire Assembly 28 ⟩ ≡

```
        while (line!=null && line.indexOf(".module ")!=0)
            line = textIn.readLine();
        if (line != null) line = line.substring (0, line.indexOf(" //"));
        ⟨ Extract (29 ".module ",30 " ",31 fileName,32 null ) 27 ⟩
    ◇
```

Macro defined by 22, 28, 33.
Macro referenced in 47.

Having thus read the Assembly-name & the associated module-name, we proceed to process the contents of the assembly. This is rather easily done:

⟨ Collect Entire Assembly 33 ⟩ ≡

```
        do {
            line = textIn.readLine();
            if (line != null  &&  line.indexOf(".namespace") >= 0)
                readNameSpace(line);
            if (line == null  ||  line.indexOf(".assembly") >= 0) {
                ⟨ Statistics at Assembly-close 19 ⟩
                return line;
            }
        } while (line != null);
        return null;
    }◇
```

Macro defined by 22, 28, 33.
Macro referenced in 47.

5. **.namespace Name**: followed by an open brace on the next line. This 'opens' the set of definitions & declarations that compose the namespace. The 'name' is read & retained. Before we detect the matching closing brace, we could find many 'class' objects, described below.

⟨ Collect NameSpace 34 ⟩ ≡

```
    static void readNameSpace (String line)
        throws IOException {
        ⟨ Extract (35 ".namespace ",36 " ",37 nameSpace ) 27 ⟩
        while ((line=textIn.readLine()) != null  &&  line.indexOf("}") != 0)
            if (line.indexOf("  .class ") == 0)
                ReadClass (line, nameSpace);
        ⟨ Statistics at Namespace-close 18 ⟩
    }◇
```

Macro referenced in 47.

6. **.class ... Name**: This is an example of an (incomplete) begining of a 'class' definition. It is typically followed by another line that names the class that this one 'Extends'. (The 'Name' is to be read and retained, and all the descriptors between the '.class' and the 'Name' are of-course subjective). Optionally there may be another line that lists the set of interfaces that this class 'Implements'. Finally after all this there is an opening brace that completes the 'start' of the class definition. Before we detect the matching closing brace, we could find many 'method' objects, described below.

   There is an interesting observation, that needs to be catered to: that Class-definitions can be nested, any depth. We therefore need to have the class-name retained in a variable that gets instanciated every-time we detect the '.class' declaration. We do this through the use of recursion. We use a *className* local variable inside a member-function definition that gets invoked when necessary.

⟨ Process Disassembled Class 38 ⟩ ≡

```
    static public void ReadClass (String line, String pref)
        throws IOException {
        String className = null, methodName = null, parentName = null;
        ⟨ Extract (39 "  .class ",40 " ",41 className ) 27 ⟩
        while ((line=textIn.readLine()) != null  &&  line.indexOf("  }") != 0)  {
            if (line.indexOf("  .class ") == 0)
                ReadClass (line, pref + "." + className);
            else if (line.indexOf("    .method ") == 0) {
                ⟨ Process Methods 42 ⟩
            }
        }
        ⟨ Statistics at Class-close 17 ⟩
    }◇
```

Macro referenced in 47.

7. **.method ... [n] ... Name (*arg-list*)**: This is an example of an (incomplete) begining of a 'method' object. The 'Name' is to be read and retained, and all the descriptors between the '.method' and the 'Name' are of-course subjective (They may include a 'newline', as well.). Finally after all this there is an opening brace that completes the 'start' of the method definition. Before we detect the matching closing brace, we could find 'call' objects, described below.

⟨Process Methods 42⟩ ≡

```
while (line!=null && line.indexOf(".method") < 0)
    line = textIn.readLine();
if (line != null)  line += textIn.readLine();
// System.out.println("Reading ... methodName: " + line);
if (line.indexOf(" (") >= 0)
    line = line.substring (0, line.indexOf(" ("));
⟨Extract (43 ".method ",44 " ",45 methodName ) 27⟩
while ((line=textIn.readLine()) != null  &&  line.indexOf("    }") != 0)  {
    ⟨Process Calls 46⟩
    // System.out.println ("Processed: " + line);
}
⟨Statistics at Method-close 16⟩
◇
```

Macro referenced in 38.

8. ***label*: call ... *fqClassName::methodName(arg-list)***: In this call-statement the label is some string, followed by a colon. *fqClassName* is the fully-qualified class-name: We are particularly interested in statments where it has the substring "System.Threading.Thread", apart from the discussion regarding 'lock's under section 2, above.

⟨Process Calls 46⟩ ≡

```
if (line.indexOf("call ") > 0  &&
    line.indexOf("System.Threading") > 0) {
    int closeS = line.indexOf("]"), openP = line.indexOf("(");
    // System.out.println ("Line: " + line + ":" + closeS + ":" + openP + ".");
    String called = openP>closeS ? line.substring(closeS+1, openP) : null;
    if (called != null) {
        mUses++;
        if (called.indexOf("System.Threading.Monitor::Enter") == 0) {
            lopens++; mUses--;
        } else if (lopens > 0  &&
            called.indexOf("System.Threading.Monitor::Exit")==0) {
            lopens--; mLocks++; mUses--;
        }
    }
}◇
```

Macro referenced in 42.

2.2.5 Building the Whole

The tool 'LockNdThread' is then composed from the above program-parts as follows:

"LockNdThread.java" 47 ≡

```
import java.io.*;
import java.util.*;
public class LockNdThread {
    static BufferedReader textIn  = null;
    static String assemblyName, fileName, nameSpace;
    static char   conLevelAt = 'a';
    static String[] t = null; // Temporary used in many places.
    ⟨Statistics Collecting State 15⟩
    ⟨Fork Disassembler, prepare to read text-MSIL 13⟩
    ⟨Collect Entire Assembly 22, ... ⟩
    ⟨Collect NameSpace 34⟩
    ⟨Process Disassembled Class 38⟩
    public static void main (String[] args) {
        try {
            ⟨Read Consolidation Level 14⟩
            textIn = populateTextReader (args);
            ⟨Begin processing input Text 21⟩
            while (line != null)
                line = readAssembly (line);
        } catch (Exception ex) {
            ex.printStackTrace(System.out);
        }
        ⟨Cumulative Statistics at Main-close 20⟩
    }
}◇
```

2.3 The Java Case

To enquire the size of the stated problem for a (set of) Class-file(s), below, we develop a program similar to the one we did for C# assemlies. We reuse concepts from the Usage spec. that we developed for C# program, and also most of its Statistics-collection & reporting infrastructure. For the Java case, we choose to name the program: SynchNdThread. An invocation of this program would look like:

java SynchNdThread [-c|-m] [class-file(s)-in-cwd | a-jar-file] ?

The enquiry is requested either for a set of (named) class-files, or for a (named) jar, or for the set of all J2SE library functionality available with the invoking JVM. The last form of invocation requires no parameters; that is theh default.

2.3.1 Disassemble Binaries into Text

Similar to what we did in the LockNdThread program, for this program, as well we want to disassemble the Java-byte-code into text, and read this text-rendering into a seperate (main) thread, and process it there. However, an approach identical to the previous case will not work very well for us in this case, because of the following differences:

- The 'monodis' utility that converted input MSIL into text output, operated upon either '.exe' or '.dll' binaries which are themselves composed of smaller entities containing MSIL corresponding to individual C# Class-files. Whereas 'javap', its equivalent utility in the Java-space, requires the 'classpath' variable to be specified on command-line, and the names of the various classes of interest to be specified also on the command-line. Since the length of the command-line is an installation dependent charachteristic, we would like not to depend upon its value being set to our convinience.
- Also, partly because of the above mentioned classpath-spec. requirement, we specified the correct usage to either specify a set of class-files, on command-line, or a jar-file, not both, intermixed as the argument. We do not allow the utility to be usable with a mix of the class and any jar file as parameters.

- In the LockNdThread program, we had just two threads: one to convert the MSIL to text, and the other (main-thread) to parse the text, and generate the output. Whereas in the SynchNdThread program, we will need to have three threads: one to repeatedly invoke the second thread with an argument list of some fixed length (like say 20 class-names in its invocation list), and the second thread to convert the byte-code to text, and then the third (main) thread to read the text generated by the second thread and generate the output.
- Some synchronisation is needed between the three threads: the first one starts the second, and therefore can findout the event of its death, while the third one needs to continue to read from a shared variable (of type inputStream), until it has done reading whatever is in there. Only after that has happened, can the first thread start the new invocation of the second thread that does the same thing for the next set of (20) classes from the jar/class-list, causing re-population of the inputStream variable that drives the processing in the third thread (the main thread).

We use the output of the 'jar -tvf ...' command, piped into a 'grep' for entries ending with a ".class" to fetch the class-names that he/she wants the program to use as a starting point. We accomplish this as implemented in the following function. We enlist all the classes thus selected into a Collection.

⟨ Get Class Names From the Jar-file 48 ⟩ ≡
```
    static public Collection
    getClassNamesFromJar (String jarFile)  {
        boolean onlyOne = jarFile != null;
        try {
            if (! onlyOne)  {
                String version = System.getProperty ("java.version"),
                       home    = System.getProperty ("java.home");
                if (version.startsWith ("1.6"))
                    jarFile = home + "/lib/rt.jar "  + home + "/lib/jce.jar " +
                              home + "/lib/jsse.jar " + home + "/lib/charsets.jar";
                              // home + "/lib/ext/sunpkcs11.jar " + home + "/../lib/tools.jar " +
                              // home + "/lib/ext/dnsns.jar;"; // + home + "/lib/ext/localedata.jar ";
                if (version.startsWith ("1.5"))
                    jarFile = home + "/lib/rt.jar "  + home + "/lib/jce.jar " +
                              home + "/lib/jsse.jar " + home + "/lib/charsets.jar " +
                              home + "/lib/ext/sunjce_provider.jar " + home+"/lib/ext/sunpkcs11.jar " +
                              home + "/lib/ext/dnsns.jar " + home + "/lib/ext/localedata.jar ";
                else if (version.startsWith ("1.4"))
                    jarFile = home + "/lib/rt.jar "  + home + "/lib/jce.jar " +
                              home + "/lib/jsse.jar " + home + "/lib/charsets.jar " +
                              home + "/lib/sunrsasign.jar " +
                              home + "/lib/ext/sunjce_provider.jar " + home + "/lib/ext/ldapsec.jar " +
                              home + "/lib/ext/dnsns.jar " + home + "/lib/ext/localedata.jar";
            }

            Process child = Runtime.getRuntime().exec (
                (onlyOne  ?  "./grepNDcut "  :  "./grepNDcutInLoop ") + jarFile);
            InputStream grepOut = child.getInputStream();
            InputStreamReader r = new InputStreamReader (grepOut);
            BufferedReader in   = new BufferedReader (r);

            TreeSet retval = new TreeSet();
            String line    = null;

            while ((line = in.readLine()) != null) {
                line = line.trim().replace ('/','.');
                line = line.substring (0, line.lastIndexOf(".class"));
                if (!retval.add(line))
                    System.err.println ("Dropped here: " + line);
            }

            if (child.waitFor() != 0)
                System.out.println ( "grepNDcut Failed: " + child.exitValue());

            return retval;
        } catch (Exception e) {
            System.out.println (e.toString());
        }
        return null;
    };◇
```

Macro referenced in 63, 83.

Notice in the above implementation, we invoke a shell script that is defined below. We choose to implement the functionality therein using the shell because originally it was designed imagining that some functionality similar to the system() C-library function would be available in java, but its closest equivalent in Java is the Runtime.getRuntime.exec interface. This interface forks another process/thread which must be passed arguments in specific constrained ways, making it very cumbersome for our use.

"grepNDcut" 49 ≡

```
    jar -tvf $1 | grep "\.class$" | cut -b 8- | cut --delimiter=" " -f 7;
    ◇
```

"grepNDcutInLoop" 50 ≡

```
    for i in $*
    do
    jar -tvf $i | grep "\.class$" | cut -b 8- | cut --delimiter=" " -f 7;
    done
    ◇
```

Recollect that since we allow two forms of usage: jar-file argument, or the class-files arguments, we need to be able to populate a common *c*lassesList variable, appropriately, based upon the form of usage detected. We do this as follows:

⟨ Populate classesList based upon usage 51 ⟩ ≡

```
    if (args.length==0 ||
        (args[args.length-1].indexOf(".class")<0 && args[args.length-1].indexOf(".jar")<0)) {
        // System.out.println ("./grepNDcutInLoop " + System.getProperty ("java.home"));
        classesList = getClassNamesFromJar (null);
        classesHome = null;
    } else if (args[args.length-1].indexOf(".class")<0  &&  args[args.length-1].indexOf(".jar")>=0) {
        classesList = getClassNamesFromJar (args[args.length-1]);
        classesHome = args[args.length-1];
    } else {
        classesList = new TreeSet();
        for (int k = 0; k<args.length;  k++)
            if (args[k].indexOf(".class")>=0)
                if (!classesList.add(args[k].substring(0, args[k].indexOf(".class"))))
                    System.err.println ("Dropped: " + args[k]);
    }◇
```

Macro referenced in 63, 83.

We now use the collection generated by the above functionality to repeatedly invoke another thread that uses the 'javap' utility to render the class-files into text. We specify this act an the implementation of a *run()* function, so that it could itself execute in its own thread. This thread is invoked by the main thread in the initial part of its main function, clearly.

The only thing that we need to be careful about, below, is that the *textIn* reference is a shared state between the thread that runs the function below, and the thread that executes the main function. The first thread creates *javap* threads successively that creates textual input for the main thread and channel it throug the said reference, while the *main* thread continues to consume it until it finds that the said reference has been nullified. Resetting the *textIn* reference to *null* ensures that the main loop terminates its activity, not expecting any more input from any *javap* threads that this run function below creates.

⟨ JavaP Invoker Thread 52 ⟩ ≡
```
     static final int  NoOfClassesPerJavapInvocation = 20;
     public void run()  {
         String  subList;          InputStream textOut = null;
         Process child    = null;  InputStreamReader r = null;  int done = 0, j;
         try {Iterator i = classesList.iterator();
             do {subList = "";
                 for (j = NoOfClassesPerJavapInvocation;  --j>=0 && i.hasNext();  )  {
                     String t = (String) i.next();
                     subList = subList + " " + t;
                     if (breakpoint (t))  break;
                 }
                 done += j;
                 child = Runtime.getRuntime().exec (System.getProperty ("java.home") +
                                                    "/../bin/javap -private -c -J-Xmx98m " +
                             (classesHome!=null ? ("-classpath " + classesHome) : "") + " " + subList);
                 textOut = child.getInputStream();
                 r = new InputStreamReader (textOut);
                 if (textIn==null)             textIn  = new BufferedReader (r);
                 else  synchronized (textIn) {  textIn  = new BufferedReader (r);  }

                 if (child.waitFor() != 0)
                     System.err.println ( "javap (" + subList + ") Failed: " + child.exitValue());
             } while (i.hasNext());
             synchronized (textIn) {  textIn = null;  }
         } catch (Exception e) {
             System.out.println (e.toString());
         }
     }

     public static boolean breakpoint (String fqcn)  {
         String version = System.getProperty ("java.version");
         if (fqcn == null) return true;
         if (version.startsWith ("1.6"))
             return  fqcn.startsWith ("com.sun.crypto.provider")  ||
                     fqcn.startsWith ("com.sun.corba.se.spi.transport") ||
                     fqcn.startsWith("sun.net.spi") || fqcn.startsWith("sun.net.www") ||
                     fqcn.startsWith("sun.text.resources");
         else if (version.startsWith ("1.5"))
             return  fqcn.equals("com.sun.crypto.provider.ai") ||
                     fqcn.equals("sun.security.pkcs11.wrapper.PKCS11RuntimeException") ||
                     fqcn.startsWith("sun.net.spi") || fqcn.startsWith("sun.net.www") ||
                     fqcn.startsWith("sun.text.resources");
         else if (version.startsWith ("1.4"))
             return  fqcn.equals("com.sun.crypto.provider.ai") ||
                     fqcn.startsWith("com.sun.jndi.ldap") || fqcn.startsWith("sun.text.resources") ||
                     fqcn.startsWith("sun.net.spi") || fqcn.startsWith("sun.net.www") ||
                     fqcn.equals("com.sun.security.sasl.util.SaslImpl");
          return true;
     }
     ◇
```
Macro referenced in 63, 83.

Towards the end of this report, in an appendix we reproduce evidence of certain valid usage of the javap command fetching surprising results. This bug seems to be there in all the three versions of the java distribution that we use for this work: 1.4.2, 1.5.0, and also 1.6.0. The `breakpoint` method iused above is to protect our analysis from the implications of this bug. It is defined above.

2.3.2 Data Agregation & Reporting

We firstly write the following peice of code to read the invocation options to find out what level of consolidation is requested in the collection & reporting of statistics. We store this information in a static variable of type 'char' in the class.

⟨ Read Java Consolidation Level 53 ⟩ ≡

```
    if (args.length > 0  &&  args[0].charAt(0) == '-')  {
        conLevelAt = args[0].charAt(1);
        if (conLevelAt != 'm'  &&  conLevelAt != 'c')  {
            System.out.println ("Usage: java SynchNdThread -[mc] [class-file-list | jar-file-name]");
            return;
        }
    }◇
```

Macro referenced in 63.

Below we define the 'state' that collects the information to be reported at the end of the run. We populate it incrementally as we go along. Also while we statically instantiate all the variables, we only populate those that need to be reported upon, based upon the consolidation-Level requested.

- **stPconLvlEntities**: This integer variable counts the number of entities that are detected to be *synchronized / thread Positive.* The choice of entities is made by the user through choosing the cut-off level, at which the consolidation is requested. For instance, the use of the '-c' invocation flag would cause the number of *infected* classes in the assemblies visited.
- **classes**: This integer counts the total number of classes seen in the input. Similarly, the counter **cumMethods** counts the total number of Methods visited. The variables **cumSynchs, & cumUses** count the number of uses of the *sychronized* and any *Thread.** function-call, respectively. The values of some appropriate subset of the counters discussed in this point, and the previous one are reported on the last line of the output of this tool, as is seen in the sample usages discussed earlier.
- **cMethods, cSynchs, cUses**: These counters count the occurrances of their corresponding entities, respectively, at a per-Class level. They are updated and reported on a per-Class basis (and reset), for those runs where the '-c' invocation flag is used. For the (only other) invocation flag "-m", the associated variables are **mSynchs, & mUses**.
- **synchMs, synchSMs, synchSeSeMs, synchSeMeMs, synchMeMs, synchEeMs**: These counters count respectively, the methods that have been marked with the synchronized modifier, those that have been additionally marked with the static modifier, methods that use the clean single-entry, single-exit monitors, those that use a single monitor, but with multiple exits, those that use nested monitors, and those that use an odd number of exits. In principle we could have two versions of each of these counters: a per-class one, and a cumulative one. However, we are currently interested only in the cumulative counts for these counters. So we avoid splitting hair here. Clearly, these counter-values are reported only on the 'Total' line.
- **mEnters, mExits**: These are counters that are counted up every time a *monitorenter* or a *monitorexit* byte-code is encountered, within a function-body. Thet are initialised to zero, and reset after every method-body close is encountered. **synchNeEuMs** (Methods that use synchronized(), by having No monitorEnters, but monitorExit Used) is a counter that counts up whenever we detect methods that have used *monitorexit* instructions in their implementation without also using *monitorenter*, before.

⟨ Java Statistics Collecting State 54 ⟩ ≡

```
static int stPconLvlEntities = 0, classes = 0;
static int cumMethods = 0, cMethods = 0;
static int cumUses = 0, cUses = 0, mUses = 0;
static int mEnters = 0, mExits = 0, synchNeEuMs = 0;
static int synchMs = 0, synchSMs = 0, synchSeSeMs = 0, synchSeMeMs = 0,
           synchMeMs = 0, synchOeMs = 0;
◇
```
Macro referenced in 63.

We update the state described above at the close of every method, and class definitions, as below. In case Method-level (or Class-level) reporting is requested, we report, iff there is non-zero values to report. The counters reported on the "Total:" line are updated unconditionally, and also reported unconditionally.

⟨ Statistics at Java-Method-close 55 ⟩ ≡
```
if (mEnters>0 || mExits>0) {
         if (mEnters==1 && mExits==1)  synchSeSeMs++;
    else if (mEnters==1 && mExits>1)   synchSeMeMs++;
    else if (mEnters>1  && mExits>1)   synchMeMs++;
    else if (mEnters==0 && mExits>0)   synchNeEuMs++;
    else if (mExits%2==1)              synchOeMs++;
    mEnters = 0;  mExits = 0;
}
if (conLevelAt == 'm'  &&  mUses>0)  {
    stPconLvlEntities++;
    System.out.println (methodName + "." + className + "(" + fileName + "):");
    System.out.println ("    " +mUses+ " 'Thread.*' calls.");
}  else if (conLevelAt == 'c')  {
    cUses += mUses;  cMethods++;
}
cumUses += mUses;  cumMethods++;  mUses = 0;◇
```
Macro referenced in 62.

⟨ Statistics at Java-Class-close 56 ⟩ ≡
```
if (conLevelAt != 'm'  &&  cUses>0)  {
    stPconLvlEntities++;
    if (conLevelAt == 'c') {
        System.out.println (className + "(" + fileName + "):");
        System.out.println ("    " +cUses+ " 'Thread.*' calls in " +cMethods+ " Methods.");
    }
}
cUses = 0;  cMethods = 0;  classes++;◇
```
Macro referenced in 63.

Finally we report the "Total:" line counters, as below.

⟨ Reporting cumulative Statistics at run-close 57 ⟩ ≡
```
System.out.println ("Total:");
                          System.out.print   ("    " + cumUses+ " 'Thread.*' calls in ");
     if (conLevelAt=='m') System.out.print   (stPconLvlEntities + "/" +cumMethods+ " Methods in ");
else if (conLevelAt=='c') System.out.print   (cumMethods+ " Methods, in " +stPconLvlEntities+ "/");
else                      System.out.print   (cumMethods+ " Methods, in ");
                          System.out.println (classes + " Classes.");
System.out.println ("    " + synchMs     + "                 synchronised   Methods,");
System.out.println ("    " + synchSMs    + "          static synchronised   Methods,");
System.out.println ("    " + synchSeSeMs + "          simple synchronised() Usages,");
System.out.println ("    " + synchSeMeMs + "     semi-simple synchronised() Usages,");
System.out.println ("    " + synchMeMs   + " nested/multiple synchronised() Usages.");
System.out.println ("    " + synchOeMs   + " odd-exit (hard) synchronised() Usages.");
System.out.println ("    " + synchNeEuMs + " ill-formed      synchronised() Usages.");
◇
```
Macro referenced in 63.

2.3.3 Classes inheriting from java.lang.Thread

In this section we develop the infrastructure to record the parent of a class when we process its definition. This is done throughout the run, and finally, we iterate once through the entire set of classes defined, to detect if a class inherits from the 'java.lang.Thread' class. We finally merely report the count of such classes.

To begin with, we observe that for every class defined in the input, it extends but one class that may or may-not inherit from any class seen so far. At the point of parsing a class definition, we know both the names of the super, as well as the sub-class. We create a HashMap Object, and record into it this information using the following strategy (The code that follows implements it):

1. We use the name of the sub-class (dereived class) as the key of a two-tupple that we "put" into the map. The name of its super-class is the 'value' associated with that key. As we do this for every class whose definition we find in the input, we could be building a chain of subclass-superclass mappings, that could be followed to find as to whether a given class does indeed inherit from a particular named class: in this case "java.lang.Thread".
2. Finally, after all the class-definitions have been consumed, and the time has come to spell out the findings, we iterate over the entries in the HashMap, finding for every one of them, if they do inherit from java.lang.Thread. In order to do this one needs to walk-up the inheritance chain, and see if any of the links encountered are called "java.lang.Thread". Clearly in this attempt, the program does more work than needs done, but we trade run-time-speed for development-time, and program-complexity.
3. Last, but not the least, we store the name of the Thread class in a "final" variable, called tP, (to mean threadParent). That variable is initialised to "java.lang.Thread" for the java varient of our program, and to "System.Threading.Thread" for the C# varient. That makes the module we develop below, truely generic.

⟨ Record Parent 58 ⟩ ≡
```
inheritanceMap.put (@1, @2);
◇
```
Macro referenced in 63.

⟨ Count Thread SubClasses 59 ⟩ ≡
```
public static int countThreadInheritors()  {
    int retval = 0;
    for (Iterator i = inheritanceMap.keySet().iterator();  i.hasNext();  )
        for (String cl = (String) i.next();  cl!=null;  cl = (String) inheritanceMap.get(cl))
            if (tP.equals(cl))  {
                retval++;  break;
            }
    return retval;
};◇
```
Macro referenced in 63.

Further, we observe, that both, in the textualised java-class-file-stream, as well as in the MSIL-stream, the name of the super-class appears on the defining line for a class, immediately following the token: " extends ". We detect this in the code associated with processing the class-name itself. It is there that we invoke the first of the functions above. The " extends " string must appear on all class-definitions $e$xcept the definition od the java.lang.Object definition.

2.3.4 The Control structure of the Program

In this section, we develop the code that parses the input data streem to identify the information of interest, and points in the streem where we would update our statistics appropriately.

A quick look at the disassembled output reveals a simple structure. Below we discuss it, alonside developing code to extract the required information. To begin with, we expect to receive the textual Java byte-code information through a member variable `textIn`, of type `BufferedReader` (populated by the function `run()`, developed in *JavaP Invoker Thread*).

1. **Compiled from "filename.java"**: declarations at the begining, 1-line per class. This information is used to populate the attribute: `fileName`.

   ⟨Read File-name declaration 60⟩ ≡

   ```
       if (line.indexOf("Compiled from ") >= 0)
           if (line.indexOf('\"')<0)  //System.out.println ("Problem-case:" + line);
               fileName = line.substring(line.lastIndexOf(' ')+1);
           else
               fileName = line.substring(line.indexOf('\"')+1, line.lastIndexOf('\"'));
       ◇
   ```

   Macro referenced in 63.

2. ***(possibly empty) modifier-list* class *class.name* extends ... {**: next, line. This is read and the fully-qualified class.name is retained. Every occurrance of such a line is followed by the entire class-definition followed by a closing brace (on a line by itself). We develop a function that is able to read the entire class definition.

   ⟨Read Java Class-name 61⟩ ≡

   ```
       if (line.indexOf(" extends ") > 0)  {
           int tint  = line.indexOf(" extends ");
           className = line.substring(line.indexOf(" class ")+7, tint);
           line = line.substring (tint + " extends ".length());
           int endParent = line.indexOf(" ")>0 ? line.indexOf(" ") : line.indexOf("{");
           parentName    = line.substring (0, endParent);
       } else {
           className = line.substring(line.indexOf(" class ")+7, line.indexOf("{"));
           parentName= null;
       }
       ◇
   ```

   Macro referenced in 63, 83.

3. ***(possibly empty) modifier-list* return-type *methodName*(...) ... ;**: This is an example of the begining of a method definition. It is typically followed by another line that contains merely the contents: **Code:**. Whenever the previous line is not followed by the "Code:" line, it means that the function is a "native" function with no byte-code implemenatin. From this line, the 'methodName' needs to be picked up and retained, and all the descriptors preceeding the 'return-type' are of-course subjective. The 'body' of the method starts on he line following the "Code:" line, and an empty line signals its closure. The close-brace signals the completion of the class-definition. Below is the code that processes the method. and its body.

   ⟨Process Java Methods 62⟩ ≡

   ```
       while (((line=textIn.readLine()) != null)  &&  line.indexOf("}") != 0)  {
           if (line.indexOf("(") >= 0)  {
               int lastBlankB4OpenP = line.substring(0, line.indexOf("(")).lastIndexOf(" ");
               methodName = line.substring(lastBlankB4OpenP<0 ? 0 : lastBlankB4OpenP, line.indexOf("("));
               String decoration = line.substring(0, lastBlankB4OpenP<0 ? 0 : lastBlankB4OpenP);
               if (decoration.indexOf("synchronized") >= 0)
                   if (decoration.indexOf("static") >=0)  synchSMs++;
                   else                                    synchMs++;
           }  else if (line.indexOf("Code:") >= 0)  {
               while ((line=textIn.readLine()) != null  &&  line.trim().length() != 0)
                   if ((line.indexOf("invokevirtual")>0 || line.indexOf("invokestatic")>0)  &&
                        line.indexOf("//Method java/lang/Thread")>0)     mUses++;
                   else if (line.indexOf("monitorenter")>0)              mEnters++;
   ```

```
                else if (line.indexOf("monitorexit")>0)             mExits++;
            ⟨ Statistics at Java-Method-close 55 ⟩
        }
    }
    ◇
```

Macro referenced in 63.

2.3.5 Building the Whole

The tool 'SynchNdThread' is then composed from the above program-parts as follows:

```
"SynchNdThread.java" 63 ≡
     import java.io.*;
     import java.util.*;
     public class SynchNdThread implements Runnable {
         static BufferedReader textIn  = null;
         static String         classesHome = ".";
         static Collection     classesList = null;
         static HashMap        inheritanceMap = new HashMap();
         static final String   tP = "java.lang.Thread";
         ⟨ Java Statistics Collecting State 54 ⟩
         ⟨ Get Class Names From the Jar-file 48 ⟩
         ⟨ JavaP Invoker Thread 52 ⟩
         ⟨ Count Thread SubClasses 59 ⟩
         public static void main (String[] args) {
             String fileName="", className="", methodName="", parentName="", line=null;
             char   conLevelAt = '\0';
             try {
                 ⟨ Read Java Consolidation Level 53 ⟩
                 ⟨ Populate classesList based upon usage 51 ⟩
                 new Thread(new SynchNdThread()).start();
                 Thread.currentThread().sleep (2000); // milliseconds
                 while (textIn != null)  {
                   synchronized (textIn)  {
                     while ((line=textIn.readLine()) != null)  {
                       ⟨ Read File-name declaration 60 ⟩
                       if (line.indexOf("class ")>=0  &&  line.indexOf("{")>=0)  {
                           ⟨ Read Java Class-name 61 ⟩
                           ⟨ Record Parent (64 className,65  parentName ) 58 ⟩
                           ⟨ Process Java Methods 62 ⟩
                           ⟨ Statistics at Java-Class-close 56 ⟩
                       }
                     }
                   }
                   Thread.currentThread().yield();
                 }
             } catch (Exception ex) {
                 ex.printStackTrace(System.out);
             }
             ⟨ Reporting cumulative Statistics at run-close 57 ⟩
             System.out.println ("Count of sub-Classes of " +tP+ ": " +countThreadInheritors());
         }
     }◇
```

2.4 Findings

We used the LockNdThread program to investigate different code-bases: MIP (5.1), and the C# .NET Libraries (Version 1.1). Below we tabulate the numbers:

Table 2.1: Findings from analysing C#.NET dlls of MIP 5.1, and .NET 1.1

Assenblies	Infected Assemblies	Infected NameSpaces	Infected Classes	Infected Methods	lock() usage	Thread. calls	Thread sub-classes
MIP .dll(s)	130 / 296	672 / 13811	679 / 14605	7767 / 229230	7427	951	
MIP .exe(s)	25 / 73	40 / 442	40 / 513	79 / 4628	43	71	
MIP (Total)	155 / 369	712 /14253	719 /15118	7846 / 233858	7470	1022	
C#.NET 1.1	29 / 40	424 / 6903	437 / 8560	1314 / 85692	584	1377	

Similarly we used the SynchNdThread program to investigate the Java runtime library (rt.jar) of the J2SE distribution for its different releases: J2SE 1.4.2_08, J2SE 1.5.0_03 and J2SE 1.6.XX. Below we tabulate the numbers:

Table 2.2: Findings from analysing rt.jar of the j2se1.4.2_08, j2se1.5.0_03, and j2se1.6.0

rt.jar Version	Infected Classes	Infected Methods	synch Methods	static Synch Methods	single synch() Methods	multi synch() Methods	Thread calls	Thread sub-classes	run time m:s
1.3.1_19	/ 5309	/ 39575	1225	319	887	85	423	28	1m24.34s
1.4.2_08	/ 9231	/ 63272	1504	380	1277	122	533	36	2m38.64s
1.5.0_03	/ 13109	/ 94397	2037	436	1732	176	876	41	3m16.28s
1.6.0-Beta	/ 16228	/ 114027	2197	471	1736	183	1020	45	3m50.12s

Chapter 3

A Hybrid Algorithm

In this chapter we discuss a hybrid algorithm for deadlock-detection that uses statically-detected information from intermidiate representation corresponding to Java or C# libraries, to identify the sets of API-functions that can, each, likely lead to deadlocks. The deadlock, if its exists, is reachable when two (or more) threads, each, invoke functions from any of such sets, concurrently. Having fetched these sets of functions that could likely, each, cause deadlocks, we then employ the dynamic component of the algorithm to convince ourselves that indeed certain specific sets indeed can deadlock the corresponding virtual-machine.

In the first section below, we sketch the details that clarify the hybrid nature of our algorithm[1]. The next section details the steps involved in the static analysis that fetches (what we call) the CCG (Concurrency Conflict Graph) corresponding to the investigated code-base. The last section discusses the design of a data-structure (and its associated operations), that captures the commonality between the processing required for a CCG (fetched from either Java or C#) to fetch its corresponding RCCG (Reduced CCG).

3.1 A High-level Sketch

In this algorithm, we use the static approach to compute until the graph, and then proceed to detect, first 2-cycles, and then 3-cycles, and then 4-cycles, ... that themselves do not have any embedded smaller-cardinality-cycles. We call the set of such cycles the set of potential-deadlocks.

For every member (n-cycle) of this set (of potential-deadlocks) we also compute (starting from all the n=2 cases), the associated set of *enabling criteria*. By that we mean to say, that we create all the objects that are needed to be passed as, possibly, parameters, and the objects, on which the functionally needs to be invoked, themselves, ... This is, of course, to be achieved recursively through the use of the Reflection-API.

Once all these enabling-sets are created, corresponding to the various n-cycles, we proceed to *time each individual function*. We do this by fetching the time in the main-thread, then invoking a function with the parameters it accepts, and then fetching the time again to find the difference. We do this thrice for a given function and find the maximum of them. (We do this because we want a good over-estimate of the time it takes to complete the computation associated with the function.) We do this for all the functions in all the n-cycles and record their timing information.

[1]Meta-detail: This hybrid deadlock-detection algorithm emerged from the very first technical discussion that we had between Prof. Dinesha and myself, on the 23rd of June, 2006, at his office, between 4 & 6PM. A plausible dynamic deadlock detection technique was discussed, and the steps that it would involve were briefly outlined, and then (after a mango-milk-shake), the static-analysis based approach that was already documented (one year ago) for the Java-case was discussed.

After discussing both the approaches, it was evident that the first part of the dynamic approach would fetch a large set, of functions that could potentially participate into deadlocks, which one would not know how to partition intelligently into smaller subsets, to explore further. How to proceed, however, with the exploration was clear. So also was evident, the fact that the static-analysis case got up to the point of computing a CCG from the input source, however it kind-of died there, not knowing where to go from there. After having discussed both, therefore, it was quite clear, that the correct thing to do was to use the static approach to compute until the graph, and then proceed to detect, first 2-cycles, and then 3-cycles, and then 4-cycles, ... that themselves do not have any embedded smaller-cardinality-cycles. We called the set of such cycles the set of potential-deadlocks.

Until this point all the computation in our program has been conducted in a single main thread. So we expect the program to not deadlock itself until this point.

We then do the following for each n-cycle Concurrently:

- Given an n-cycle, start m+1 threads: 1 thread each corresponding to the m public members (m <= n) of the cycle, and one last one for a timer to detect the condition that the intended deadlock has occurred.
- In this (n-cycle) case, every one of the m-threads starts executing a loop, at the beginning of which we have a m-barrier. So every member of this m-set of threads starts at the barrier together, but executes the function assigned to itself. If control returns from the call, then the next iteration awaits the arrival of all threads to the barrier. At the barrier, we also start the timer thread with a value equal to the sum of all the timing-values plus some extra. The Timer thread flags the occurrence of the deadlock if it times out before the occurrence of a reset event from the start of the next iteration. (NOTE: There are unaddressed problem cases here: What if the GC occurs causing the timer to go off first, but no deadlock having been reached? ...)
- So we have as-many barriers as there are cycles in our pottential-deadlock set, to start with. We go over each of these barriers some pre-determined number of times, deadlocking wherever possible, by the time we are done. Every deadlock detected by the timer thread is reported by it, and that set of threads end-up remaining in that local-deadlock, and finally we exit from all the un-deadlocked n-cycles , and have no work to do. The user of our program then kills the multiply-deadlocked JVM, fetching the stack-traces. These can then be reported to the SUN-Java Team. And Microsoft!! ?

3.2 The Concurrency Conflict Graph: Its Construction

In this section we take a macro-level perspective, to construct a data-structure (CCG), using static analysis of the built code-base. Recollect our observation that when a synchronized method called on one thread (eventually) leads to the call of another synchronised method that causes the thread to block, for a lock that has already been acquired by some other thread that invoked some synchronised method; and if the second thread manages to require the lock that the first thread had acquired, then we have successfully reached a deadlock.

Below we enumerate the steps to construct, starting from a cleanly built J2SE Workspace, or the .NET assemblies corresponding to the CLR (C# Language Runtime), the data-structure we call a "Concurrency Conflict Graph".

1. Step 0 - Specifying the input to the step 1 below: The input to this algorithm is the jar files produced by a successful build of the j2se-workspace, or the .NET assemblies corresponding to the CLR from Microsoft. The fact that their build has been successful implies that the source-code that had been compiled into the class-files (or MSIL files) that the jars (assemblies) comprise of is all well-formed. In the algorithms below we avoid a lot of work by relying upon this fact. So the code below is not designed to be robust in the face of neither ill-formed classes, nor those corresponding to bad java/C#.
2. Step 1 - Unconditional call-graph construction: In this step we iterate over all methods of all classes that compose the jar files (.NET assemblies) that are input to the tool. For each method we execute the steps below:
3. Step 1.1 - Create a node of the graph: We create a node corresponding to the method. The node stores the fully qualified name, called `fullName` of the method as the key peice of information, including its signature. Such a fully-specified node is inserted into a map, anong with its fully-qualified-name as the key. (While creating the node given its key, if in the collection (hashMap) of un-explored nodes we detect that this key exists, then we delete that map-entry form that list, indicating that we are now in the process of parsing the full definition of the code-blob, and use its associated value to populate it completely, and enter the key-value pair into the collection (hashMap) of fully-specified nodes). Notice, that as a part of this algorithm, we maintain two hashMaps: `unexploredCodeBlobs`, and `exploredCodeBlobs`.

4. Step 1.2: We tag the node if it is either synchronized, or static -synchronized. The name of the class is stored as an attribute. This attribute is suffixed by a ".class" IFF the method is static, in addition to being synchronized. This information is held in a field called `lockType`.

5. Step 1.3 - Populate its immediately-called-functions attribute, `calledCodeBlobs`: In this step we parse through the method-body, and for every function that is invoked (irrespective of the criteria used to invoke it and irrespective of the argument-list passed to its invocation), we insert a directed-edge between this current node, and the node corresponding to the called method. In case there is no node corresponding to the called method, we create it as per Step 1.1. We have one special case here: If there is the use of the 'monitorenter' & 'monitorexit' instructions in the method body, then for the peice of code enclosed by them (earliest monitorenter & the matching monitorexit (s?)) we achieve this step by executing as per Step 1.3.1 below.

6. Step 1.3.1 - Process functions called-under-synchronization: Corresponding to this delineated code-blob, create a new node into the original graph. For key, use a null. (Although the collection of nodes is keyed, we allow it to contain multiple nodes whose key is null. We make this exception in this otherwise single-valued set.) We do store the type of the referenced-object that is to be locked before this code-blob can be executed. The signature attribure is left blank, as well. We tag this node as being synchronised. We recursively re-enter the processing of this step whenever we encounter another 'monitorenter' statement as a part of the code-blob we are processing. Now for all functions called in this code, we create a node for each of them as in Step 1.3, except that we use this null-key node that we have just created as the 'caller' node. So by the end of this step we would have (exacly one) null-key node that we created as a part of this call. And with that as parent we may have 0 or more other nodes reachable through directed-edges from it. The return value of this peice of code is the null-key node it created. The caller of this function is expected to use that return value node, to update the graph with one directed edge that reaches into the thus returned node. The caller could be either the code in step 1.3 or step 1.3.1.

In steps 0 through 1.3.1, above, we have completed constructing a superset of the static-call-graph, without taking into acount the polymorphism feature of the C#, not Java Languages. It is easy to see why the above computation should terminate. It iterates as many times as the number of class-files found in the jars/.NET-assemblies. Every class file ought to be defined exactly once in a well-formed workspace. And its step 1.3.1 recursively calls itself only as many times as it encounters the 'monitorenter' instruction, (which is again finite). So the computation above does terminate.

There remains another final step in the algorithm.

7. Step 2 - Prune the graph off all non-blocking code-representation: At the end of this step, we would like the graph to have exacly only those nodes that represent synchronized methods, static-synchronized methods, and such methods that use the *synchronized() / lock()* block in their implementation. However, while we do this, we also augment the semantics associated with the directed-edge in this graph. In the previous step (Step 1) of this algorithm, we used the directed-edge to mean a *direct* potential call at run-time, by the from-method, of the to-method. After this step (Step 2), a directed-edge in the graph will mean an *direct or eventual* pottential call, with possibly other non-blocking method calls in the call-stack, in between the stack-frame corresponding to the from-method, and that corresponding to the to-method. In two sub-steps we indicate how this can be achieved with least effort.

8. Old Step 2.1.1 - Form a set of non-synchronized-nodes reachable from any synchronized node: To begin with, form a set of all such non-synchronized nodes that are directly reachable from any synchronized-node. Then compute the transitive-closure of this set, adding to it exacly such nodes that are reachable from some node already in it, and iff it is non-synchronized.

9. New Step 2.1: In this step, update the nodes from the `ccgNodes` by changing their `calledCodeBlobs` attribute to identify (instead all) those other members from that (very) restricted set (of `ccgNodes`) which are reachable from itself. In view of this (new) step, the older steps 2.1.1 & 2.1.2 are now not necessary, and therefore the `inCount` member, and code updating or using it may be safely removed.

Moreover, the calledCodeBlobs now needs to become an object of type Set, rather than type List. We make these changes, in the code above.

10. Old Step 2.1.2 - Desolve the above node-set, one-by-one, while importing the information about reachability through them into the rest of the graph: For every node i in the above set, do the following: For every member j of the set of nodes that have a directed-edge between i to j, add (if not already present, and only when $((i \mathrel{!}= j)\ \&\&\ (j \mathrel{!}= k)))$ a directed-edge from i to every node k, where k iterates over the set of nodes directly reachable from node j through a directed-edge. Having done this, delete the edge between the node i, to node j. Also, if it so happens that the just deleted edge was the last incomming edge into the node j, then that node, along with all the out-going edges from it, could also be deleted. However, with the book-keeping information that we have been maintaining so far, we cannot wasily determine if the edge to be deleted is the last one pointing into the node it points to. We therefore, add a reference counter to the nodes, and increment, and decrement it, everytime we add or remove an edge that points into it. We call this member the `inCount`.

It is easy to see that the initial-set cardinality for step 2.1.1 is finite. And its transitive-closure *w.r.t.* reachability can grow maximum to some larger subset of the node-set of the graph. So it is easy to see that that computation is a terminating computation.

Similarly for Step 2.1.2, we can argue that it also is a finite computation. More interestingly, it is noteworthy, the manner in which this step is oblivious of cycles in the subgraph represented by the set constructed in Step 2.1.1. Since it adds directed-edges only between those nodes that are distinct from itself, it effectively shrinks cycles as the computation proceeds. The cycles shrink to the point that they end up being self loops on nodes. Since all nodes in the subset, would be eliminated by the end of that step, the cycles get quite eligently handled.

10. Step 2.1.3 - Unconditionally drop rest of the representatives of non-synchronized conputation: In the above two (sub)steps, we have handled all non-synchronized nodes that are reachable from some synchronized node. And they are already removed from the graph. There would be a lot of nodes that do not fall into the above category. Since they do not contribute in any way to better our understanding of the deadlocking sceanarios, we drop them from the graph. More precisely, we delete from the graph, all such nodes that do not represent synchronized-methods, nor static-synchronized methods, nor those that use the synchronized statement-block in the implementation.

11. Step 2.2.1 - For *every* method, that uses the synchronized-block, drop all the other code representation: Recollect that the way we representated this aspect of the code was through the use of a named-node (corresponding to the method) that had an outgoing directed-edge into an unnamed-node (that represented the synchronized-block). There could have been other outgoing directed-edges from the named-node. Drop them, unless they reach synchronized methods/code-blocks. If the reached method is a synchronized method (say k), then insert a directed-edge (each) from all such methods that can reach *this*-method to k. Then drop the directed-edge from *this*-node to k. Do this for all named-synchronized-nodes k directly reachable from *this*-node.

12. Step 2.2.2 - Effectively 'inline' synchronized-code-blocks: Once Step 2.2.1 is done, there will be only such unnamed-nodes reachable from *this*-node that are corresponding to the use of (seperate and/or possibly-nested) synchronized-statement-blocks in the code for this method. Give all of them (distinct) names derieved (possibly by suffixing an up-counting-integral-value) from the name of *this*-method. Now, *desolve* this-node, given that it is now reaching all (newly-)named nodes directly. (This can be done in a manner very similar to the description in step 2.2.1.)

At the end of the algorithm, we have nodes in the CCG (`ccgNodes`) that all correspond to the act of firstly acquiring a lock (or having to block on it), doing some useful work, and then finally releasing the lock. The directed-edges `ccgEdges` represent the *possibility* that the thread tries to acquire *another* lock, after already been holding on to the ones acquired before. Of course, the lock could be either on an *object*, or on a *class*. And there are lots of objects and classes in the system.

3.3 Data-Structure Design

It is clear from the discussion of the previous section that we have at hand a graph structure, where the nodes are (not necessariely) keyed, and they have attributes, and the edges between them are directed. Here we detail the java-implementation of a "CodeBlob" class, and we use a hash-tree to store objects of this type. These objects represent an actual code-blob that either does or doesnot hold some lock while it is calling other methods, if any. Also such code-blobs might-not have a name; but if they do, then it represents the entire body of the corresponding java-method. The purpose of this data-structure is to capture the commanality between the implementations that handle the investigation for Java & C#.

```
"CodeBlob.java" 66 ≡
     ⟨ JavaFile Opener 84 ⟩
     public class CodeBlob {
         String  lockType;
                // Stores the type of object that is synchronised upon:
                //  Null ==> Object represents an 'un-Synchronised' code-blob
                //  Add a ".class" postfix for static methods after package-name.class-name
                //  Necessariely maps into the locks hashMap and updates it whenever set/reset.

         String  fullName;   // "package-name/class-name.method-name:signature"
         boolean isPublic;   // Is this a publicly accessable method ?
         int  inCount;       // Counts the number of references to this node, through the code-base.

         static int natives = 0;     // counts native methods noticed.

         HashSet calledCodeBlobs;    // references "called" code-blobs.
         HashSet alternateCodeBlobs; // references "alternate" code-blobs that 'could' handle the call.

         static HashSet          interfaces  = new HashSet (); // lists all interfaces seen.
         static HashMap               locks  = new HashMap (); // Maps locks->codeBlobs that block on it
         static HashMap      inheritanceMap  = new HashMap ();
         static HashMap   exploredCodeBlobs  = new HashMap (); // Code-blobs seen.
         static HashMap unexploredCodeBlobs  = new HashMap (); // Code-blobs refered in those seen.
         static Set         abstractClasses  = new HashSet (); // Collects names of Abstract classes
         static Set     ccgNodes             = new HashSet (); // The Concurrency-conflict-graph.
                Set     ccgEdges;

     ◇
```

File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

Looking back at 'Step 1.1' of the algorithm above, we would need a constructor for the CodeBlob class that accepts 6 parameters: Class-name, Method-name, signature, is public?, is static?, is synchronized?, and is Native?.

```
"CodeBlob.java" 67 ≡
         CodeBlob (String fsClassName, String methodName,
             String signature, boolean isPub, boolean isStatic, boolean isSynch, boolean isNative)  {
             fullName = fsClassName + "." + methodName + ":" + signature;
             setLockType (isSynch ? (fsClassName + (isStatic ? ".class" : "")) : null);
             isPublic = isPub;    calledCodeBlobs    = null;
             ccgEdges = null;     alternateCodeBlobs = null;        inCount = 0;
             natives += (isNative ? 1 : 0);
             // System.out.println ("New xplored Node: " + fullName);
         }
     ◇
```

File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

However, since we would like not to have more than one nodes attempting to represent the came code-blob, we retain the above method as non-public, and provide the following static method as the one that users would be able to invoke.

```
"CodeBlob.java" 68 ≡
         public static CodeBlob newCodeBlob (String fsClassNm, String mNm,
             String prtEncoding, boolean isPub, boolean isStatic, boolean isSynch, boolean isNative)  {
             CodeBlob rv = (CodeBlob) unexploredCodeBlobs.get(fsClassNm +"."+ mNm +":"+ prtEncoding);
             if (rv == null)
                 rv = new CodeBlob(fsClassNm, mNm, prtEncoding, isPub, isStatic, isSynch, isNative);
             else  {
                 rv.setLockType (isSynch ? (fsClassNm + (isStatic ? ".class" : "")) : null);
                 rv.isPublic = isPub;
                 natives += (isNative ? 1 : 0);
                 unexploredCodeBlobs.remove (rv.fullName);
             }
             exploredCodeBlobs.put (rv.fullName, rv);
             if (rv.getLockType() != null)
                 ccgNodes.add (rv);
             return rv;
         }
     ◇
```

File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

In order to take note of every call to some code-blob, as specified in 'Step 1.3', we define the following function. In this function we check if the `calledCodeBlobs` is initialised, since all the constructors for this class set it to null. Moreover notice that any attempt to use the function below to register calls from a codeBlob to itself cause no effect. The function below merely returns, without creating any self-loops in the graph. Finally, after registering the call, we increment the `inCount` of the *called* codeBlob. The code-fragment corresponding to 'callability from Step 1.3.1' is discussed a little later.

```
"CodeBlob.java" 69 ≡
         public CodeBlob noteCallTo (String fsMethodSignature)  {
             if (fsMethodSignature != null  &&  fullName.equals(fsMethodSignature))
                 return null;
             if (fsMethodSignature == null)
                 fsMethodSignature = getUniqueName();
             CodeBlob rv = (CodeBlob) exploredCodeBlobs.get (fsMethodSignature);
             if (rv==null  &&  (rv = (CodeBlob) unexploredCodeBlobs.get (fsMethodSignature))==null)
                 unexploredCodeBlobs.put (fsMethodSignature, rv = new CodeBlob(fsMethodSignature));
             if (calledCodeBlobs == null)
                 calledCodeBlobs = new HashSet ();
             calledCodeBlobs.add (rv);
             return rv;
         }
     ◇
```

File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

We need to define the constructor that we have used above. This one is also retained non-public.

```
"CodeBlob.java" 70 ≡
         protected CodeBlob (String fullName)  {
             this.fullName      = fullName;    setLockType(null);  isPublic = false;
             calledCodeBlobs    = null;        ccgEdges = null;    alternateCodeBlobs = null;
             // System.out.println ("New un-xplored Node: " + fullName);
         };
     ◇
```

File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

noteCallTo can also be used to process the Step 1.3.1. However, we need to take some special care of the fact that when called in this manner, the methodName nor the signature are really available. At the point of call, therefore, lets say, that all the parameters are null. In the code-fragment below, we generate a unique name that is assigned to fullName in this special case, before proceeding to process further, in the method: noteCallTo.

```
"CodeBlob.java" 71 ≡
         static int j = 0;
         protected static String getUniqueName ()  {  return "temp" + j++;  }
     ◇
```
File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

Another method that this class ought to export is the following. Using it, the code under Step 1.3.1 can set the type of the object that the *monitorEnter* statement is issued upon, as the lockType. Ideally, as to whether the object that is attempted to be locked is static, is also relevant information that needs to be recorded.

```
"CodeBlob.java" 72 ≡
         public void setLockType (String lock)  {
             setLockType (lock, false);
         }
         public String getLockType ()  {    return lockType;       }
         public void setLockType (String lType, boolean isStatic)  {
             String old    = getLockType();
             lockType = lType==null ? null : (lType + (isStatic ? ".class" : ""));
             if (old != null)
                 ((HashSet)locks.get(old)).remove (this);
             if (getLockType() != null)
                  if (locks.containsKey(getLockType()))
                      ((HashSet)locks.get(getLockType())).add(this);
                  else  {
                      HashSet why = new HashSet();
                      why.add (this);
                      locks.put(getLockType(), why);
                  }
                  if (old == null  &&  getLockType() != null)  ccgNodes.add (this);
             else if (old != null  &&  getLockType() == null)  ccgNodes.remove (this);
         }
     ◇
```
File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

Step 1.4 requires us to update the state of nodes corresponding to such CodeBlobs that are inherited from super-classes, and happen to called into. Until we do this then the edge corresponding to such a call leads from the caller into a CodeBlob object that hangs off the unexploredCodeBlobs list. Whereas it ought to refer to some CodeBlob object that is well-defined. Below, we use the information in the inheritanceMap to match-up such CodeBlob objects and *uprade* them from being undefined to being well-defined.

```
"CodeBlob.java" 73 ≡
         public static void defineReferencedInheritedFunctions ()  {
             HashSet removables = new HashSet ();
             for (Iterator i = unexploredCodeBlobs.keySet().iterator();  i.hasNext();  )  {
                 String sign, clNm, rest;
                 sign = (String) i.next();
                 if (sign == null  ||  sign.indexOf(".") < 0  ||  sign.indexOf(":") < 0)
                     continue;

                 clNm = sign.substring (0, sign.indexOf("."));
                 rest = sign.substring (sign.indexOf("."));
```

```
            // System.out.println ("clNm = " + clNm + "\nrest = " + rest);
            for (String p = (String) inheritanceMap.get(clNm);  p!=null;
                        p = (String) inheritanceMap.get(p))  {
                CodeBlob j, k;
                if ((j = (CodeBlob) exploredCodeBlobs.get (p + rest)) != null)  {
                    if ((k = (CodeBlob) unexploredCodeBlobs.get(sign)) != null)  {
                        k.inheritDefinition (j);
                        removables.add (sign);
                        exploredCodeBlobs.put (k.fullName, k);
                    }  else
                        System.out.println ("Found <" +sign+ ", null> in unexploredCodeBlobs!");
                    break;
                }
            }
        }
        if (! removables.isEmpty())
            for (Iterator i = removables.iterator();  i.hasNext();  )
                unexploredCodeBlobs.remove ((String) i.next());
        System.out.println ("Upgraded " + removables.size() + " nodes into explored-CodeBlobs.");
    }

    public void inheritDefinition (CodeBlob pM) {
        // System.out.println ("Creating " + fullName + " from " + pM.fullName);
        setLockType (pM.getLockType());
        isPublic = pM.isPublic;
        calledCodeBlobs = pM.calledCodeBlobs == null  ?  null  :  new HashSet (pM.calledCodeBlobs);
        alternateCodeBlobs = pM.alternateCodeBlobs == null  ?  null
                                                            :  new HashSet (pM.alternateCodeBlobs);
    }
◇
```

File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

Step 2.1 requires us to identify the set of all nodes that correspond to 'synchronized' code-blobs and then locate all the reachable synchronized-code-blobs from those. Clearly, the set of all 'synchronized' code-blobs would be in the `exploredCodeBlobs` set. So rather than iterating through the entire set to locate them, which may not be efficient, notice that we have another static member in that incrementally book-keeps this information so that it is readily available for our purpose: `ccgNodes`. In the code-fragment below, we recursively compute the required transitive closure to minimise the unnecessary work done.

```
"CodeBlob.java" 74 ≡
    public static void SynchCodeReachedFromSynchCode ()  {
        for (Iterator i = ccgNodes.iterator();  i.hasNext();  )  {
            CodeBlob n = (CodeBlob) i.next();
            if (n.calledCodeBlobs != null)  {
                Set rv = new HashSet ();
                n.augmentReachableSynchNodeSet (rv, new HashSet(),
                                               n.getLockType() + " " + n.fullName);
                n.ccgEdges = rv;
            }
        }
    }

    void augmentReachableSynchNodeSet (Set rv, Set workArea, String callStack)  {
        if (calledCodeBlobs != null)
            for (Iterator j = calledCodeBlobs.iterator();  j.hasNext();  )  {
                CodeBlob m = (CodeBlob) j.next();
                if (m.getLockType() != null)
                    rv.add ((callStack + " " + m.fullName + " " + m.getLockType()).trim());
                else if (workArea.add (m))
                    m.augmentReachableSynchNodeSet (rv, workArea, callStack + " " + m.fullName);
                if (m.alternateCodeBlobs != null)
                    for (Iterator k = m.alternateCodeBlobs.iterator();  k.hasNext();  )  {
                        CodeBlob n = (CodeBlob) j.next();
                        if (n.getLockType() != null)
                            rv.add ((callStack + " " + n.fullName + " " + n.getLockType()).trim());
                        else if (workArea.add (n))
                            n.augmentReachableSynchNodeSet (rv, workArea,
                                                            callStack + " " + n.fullName);
                    }
            }
    }
◇
```

File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

In view of the above implementation of the New Step 2.1, we do not need to provide any implementation corresponding to the old Steps 2.1.1, and 2.1.2.

The implementation for Step 2.1.3, which is to drop all non-synchronized nodes from the computation is rather simply done: with the help of the garbage collector. We merely nullify the collections assigned to the `unexploredCodeBlobs`, and the `exploredCodeBlobs`, and invoke `System.gc()`.

```
"CodeBlob.java" 75 ≡
        public static void dropUnsynchronizedCodeBlobs ()  {
            System.out.println ("Native method count (definitions not seen): " + natives + ".\n" +
                "Non-Synch method count: " + (exploredCodeBlobs.size() - ccgNodes.size()) + ".\n" +
                "Synch-Code-blobs count: " + ccgNodes.size() + ".");
            exploredCodeBlobs = null;
            System.out.println ("The following are undefined functions:");
            Object[] undefineds = unexploredCodeBlobs.values().toArray();
            int j = 0;
            for (int i = undefineds.length;  --i >= 0;  )
                if (! ((CodeBlob)undefineds[i]).fullName.startsWith("temp"))  {
                    int    iDot = ((CodeBlob)undefineds[i]).fullName.indexOf(".");
                    if (iDot < 0)  continue;
                    String clNm = ((CodeBlob)undefineds[i]).fullName.substring (0, iDot);
                    if (abstractClasses.contains (clNm))  continue;
                    System.out.println ("undefined: " + ((CodeBlob)undefineds[i]).fullName);
                    j++;
                }
            System.out.println ("Undefined method count: " + j + ".\n");
            unexploredCodeBlobs = null;
            System.gc();
        };
    ◇
```

File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

Finally, before we get done, we need to be able to detect and report cycles in the CCG fetched from the above computation, that all represent possible deadlocks in the analysed source. In order to do this we proceed below to develop a function that would report cycles of a given length. It is important to define what we mwan by lock equality before getting into the details of the implementation below:

Two objects of type `Codeblob` are defined to be pottentially in conflict, iff:

1. the `lockType` attribute for both of them are equal, **and** they both end with a `.class`.
2. the `lockType` attribute for *neither* of them ends with a `.class`, and, one of them names a class that is either an ancestor, or a decendent of that named by the other.

In order to provide for our ability to detect the case 2, above, we use the information recorded in the `inheritanceMap`, namely the superClass for all the classes that we see during our analysis: similar to the way we did it in the previous chapter, to detect if a class was inheriting from the System.Threading.Thread class.

"CodeBlob.java" 76 ≡
```
        public boolean canLockBeTheSame (CodeBlob x)  {
            if (x == null  ||  x.getLockType() == null  ||  getLockType() == null)
                return false;
            else if (x.getLockType().endsWith(".class")  &&  getLockType().endsWith(".class")  &&
                     getLockType().equals(x.getLockType()))
                     return true;
            else  {
                String a = getLockType(), b = x.getLockType(), t;
                if (a.endsWith(".class"))  a = a.substring(0, a.lastIndexOf("."));
                if (b.endsWith(".class"))  b = b.substring(0, b.lastIndexOf("."));
                for (t = a;  t != null;  t = (String) inheritanceMap.get(t))
                    if (b.equals(t))   return true;
                for (t = b;  t != null;  t = (String) inheritanceMap.get(t))
                    if (a.equals(t))   return true;
                return false;
            }
        };
```
◇

File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

We now get to the point where we seem enabled to develop the code-fragment that shall loop over the CCG reporting all cycles, begining with all the 2-cycles, then 3-cycles, and so on. In the function `findCycles` below, there are two parts: the then-part, and the else-part of the null-check of its first parameter. The then-part starts off the recurrsion that has a base case in the else-part. Every initial invocation of this function looks for cycles of definite length. So for instance when *finding* a 5-cycle, the *sixth* node is necessariely identical to the *first* node in the cycle. And the `lNodes` stores the first 5 nodes along the path. The value of `cLen` is decremented for every subsequent recursive call; for the initial call, it is used to indicate to the cycle-detector functionality as to the maximum `n` for which `n`-cycles are asked to be detected. Therefore notice that in the *then-part* below, the value of `cLen` is used to terminate the for-loop.

"CodeBlob.java" 77 ≡
```
        static TreeSet t12hc = new TreeSet(), lhcs = new TreeSet();
        public static void report2LockDeadlockingThreads ()  {
            HashSet donePairs = new HashSet();
            int    lhc;                  // lock  Hash-Code, where l1 is less than l2, always.
            TreeSet t1hc = new TreeSet(),
                    t2hc = new TreeSet(); // Thread-1, and Thread-2 Hash-Codes, respectively.
            for (Iterator li = locks.keySet().iterator();  li.hasNext();  )  {
                String    l1 = (String)  li.next();
                HashSet cbl1 = (HashSet) locks.get(l1);
                for (Iterator lri = locks.keySet().iterator();  lri.hasNext();  )  {
                    String    l2 = (String)  lri.next();
                    HashSet cbl2 = (HashSet) locks.get(l2);
                    if (l1.equals(l2)  ||  donePairs.contains (l1 +"-"+ l2)  ||
                        !reachableFromTo (l1, l2)  ||  !reachableFromTo (l2, l1))
                        continue;
                    System.out.println ("<2-Cycle " + l1 +" "+ l2 + ">");
                    lhc   =   l1.compareTo(l2) <= 0  ?  (l1 +"-"+ l2).hashCode()
                                                 :  (l2 +"-"+ l1).hashCode();
                    for (Iterator cbi = cbl1.iterator();  cbi.hasNext();  )  {
                        CodeBlob cb = (CodeBlob) cbi.next();
                        if (cb.ccgEdges == null)  continue;
                        for (Iterator ccgEi = cb.ccgEdges.iterator();  ccgEi.hasNext();  )  {
                            String ccgE = (String) ccgEi.next();
                            int lb = ccgE.lastIndexOf (" ");
                            if (lb >= 0  &&  ccgE.substring(lb+1).equals(l2))  {
                                System.out.println ("    <Thread-1 Option>");
                                System.out.println (
                                    ccgE.substring(ccgE.indexOf(" ")+1, lb).replace(' ', '\n'));
                                System.out.println ("    </Thread-1 Option>\n");
                                t1hc.add(new Integer(ccgE.substring(ccgE.indexOf(" ")+1, lb).hashCode()));
                            }
                        }
                    }
                    for (Iterator cbj = cbl2.iterator();  cbj.hasNext();  )  {
                        CodeBlob cb = (CodeBlob) cbj.next();
                        if (cb.ccgEdges == null)  continue;
                        for (Iterator ccgEj = cb.ccgEdges.iterator();  ccgEj.hasNext();  )  {
                            String ccgE = (String) ccgEj.next();
                            int lb = ccgE.lastIndexOf (" ");
                            if (lb >= 0  &&  ccgE.substring(lb+1).equals(l1))  {
                                System.out.println ("    <Thread-2 Option>");
                                System.out.println (
                                    ccgE.substring(ccgE.indexOf(" ")+1, lb).replace(' ', '\n'));
                                System.out.println ("    </Thread-2 Option>\n");
                                t2hc.add(new Integer(ccgE.substring(ccgE.indexOf(" ")+1, lb).hashCode()));
                            }
                        }
                    }
                    System.out.println ("</2-Cycle " + l1 +" "+ l2 + ">\n\n");
                    String t1s = "", t2s = "";
                    for (Iterator i = t1hc.iterator();  i.hasNext();  )   t1s += "=" + i.next();
                    for (Iterator j = t2hc.iterator();  j.hasNext();  )   t2s += "=" + j.next();
                    System.out.println ("HC:" + lhc + "<--" + ((l1.compareTo(l2) <= 0) ? t1s : t2s));
                    System.out.println ("HC:" + lhc + ">--" + ((l1.compareTo(l2) <= 0) ? t2s : t1s));
                    lhcs.add (new Integer(lhc));
                    t12hc.addAll (t1hc);  t12hc.addAll (t2hc);
                    lhc = 0;  t1hc.clear();  t2hc.clear();  t1s = "";  t2s = "";
                    donePairs.add (l1 +"-"+ l2);
                    donePairs.add (l2 +"-"+ l1);
                }
            }
        }
        public static boolean reachableFromTo (String l1, String l2)  {
            HashSet cbl1 = (HashSet) locks.get(l1);
            for (Iterator cbi = cbl1.iterator();  cbi.hasNext();  )  {
                CodeBlob cb = (CodeBlob) cbi.next();
                if (cb.ccgEdges == null)  continue;
                for (Iterator ccgEi = cb.ccgEdges.iterator();  ccgEi.hasNext();  )  {
                     String ccgE = (String) ccgEi.next();
                     int lb = ccgE.lastIndexOf (" ");
                     if (lb >= 0  &&  ccgE.substring(lb+1).equals(l2)) return true;
                }
            }
            return false;
        }
        public static void findCycles (CodeBlob[] lNodes, Set oEdges, int cLen)  {
            if (lNodes == null)
                for (int i = 2;  i <= cLen;  i++)  {
                    lNodes = new CodeBlob[i+1];
```

```
                    for (Iterator ccgi = ccgNodes.iterator();  ccgi.hasNext();  )  {
                        lNodes[i] = (CodeBlob) ccgi.next();
                        if (lNodes[i].fullName.startsWith("temp"))    continue;
                        findCycles (lNodes, lNodes[i].ccgEdges, i-1);
                    }
                }
            else if (oEdges != null)
                for (Iterator ei = oEdges.iterator();  ei.hasNext();  )  {
                    CodeBlob cb = (CodeBlob) ei.next();
                    boolean embeddedCycle = false;
                    for (int i = lNodes.length-1;  cLen>0 && i>cLen;  --i)
                         embeddedCycle = embeddedCycle || lNodes[i].canLockBeTheSame (cb);
                    if (embeddedCycle)
                        continue;
                    lNodes[cLen] = cb;
                    if (cLen != 0)
                        findCycles (lNodes, cb.ccgEdges, cLen-1);
                    else if (lNodes[lNodes.length-1].canLockBeTheSame (cb))
                        PrintCycle (lNodes);
                }
        };
    ◇
```
File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

The `PrintCycle` method prints the specifics of the CodeBlobs along the cycle. A more useful implementation could also try reporting all the synch *and non-synchronized* nodes along the path that could pottentially realise the cycle. The basic information is printed by the implementation below:

```
"CodeBlob.java" 78 ≡
        public static void PrintCycle (CodeBlob[] lNodes)  {
            int n = lNodes.length;
            System.out.println ("<" + (n-1) + "-cycle>");
            for (int t = n;  --t >= 0;  )
                System.out.println ("    " + lNodes[t].fullName + "\n\tlocks " +
                                            lNodes[t].getLockType() + " and calls ");
            System.out.println ("</" + (n-1) + "-cycle>\n\n");
        }
    }◇
```
File defined by 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78.

Notice that with the above function definition, we have also closed the definition of the class.

3.3.1 Discuss

Consider the CCG in the context of the test-case we discussed in the previous section. The deadlock that was reached therein would correspond to two paths in the CCG. One of the nodes in the first path would correspond to the "java.util.TimeZone.getTimeZone". There would be a directed-edge from that node to another corresponding to "sun.util.calendar.ZoneInfoFile.getZoneInfo". And from there to another node corresponding to "java.util.Properties.getProperty".

Similarly in another path of the CCG there would also be a node corresponding to "java.util.Properties.store" that would have an outgoing directed-edge to the node corresponding to "java.util.TimeZone.getDefault".

Notice that these two paths are not part of any cycle structure, but there are nodes along both the paths that corresponding to static synchronized methods of the same class. And another commanality is the fact that there are nodes along both the paths that correspond to synchronized methods of the "java.util.Properties" class. And these two types of nodes occur in inverted order along the two paths, and when these two threads are invoked *concurrently* with the same properties Object, all the requirements for reaching the deadlock are met.

3.3.2 Deficiencies that need rectification

Upon geting the first-cut implementation done, the following specific improvement opportunities were still available to pursue.

1. "clone()" called on array objects. 174 instances in 1.6.0, and 88 instances in 1.5.0.
2. Certain jars reachable through the javap command are not parsed as a part of the analysis.
3. Unspecified methods of abstract classes are reported as not-found.
    (a) One way to handle this is to treat them the same way that we treat invocation of interface-methods: ignore them.
    (b) The correct way to handle this is to create the inheritance tree below the point of undefined-function, and to identify methods of all sub-types that could get invoked at the point of the original call.
4. The approach in 3-b can be taken for all interface methods, as well, and therby make the graph more complete.
5. In either case, there needs to be a way to report cycles that is more concise than as they are reported currently. What is that way ? This is to be done as follows:
    - Firstly create shared Lock objects corresponding to the various locks ever aquired by any code in the library.
    - Then, connect these lock-objects using annotated arcs, whenever there is call-sequence in the code-base wherein a public-method aquiring the first lock can call into functionality that can eventually require to acquire the other lock. The annotation is an ordered set of methods starting with the public-method, and ending with the method that aquires the destination lock. Report 2-thread, 2-lock deadlocks in XML Notation, in entirity.
6. Orthogonally, there is a need to do symbolic-interpretation at the type-level to identify the lock-type of locks associated with partial code-blobs. This can be done.
7. The design & implementation of 5 has to be handled more scientifically: The annotation needs to be better-thought:
    - The first method in the ordered list need not be a public method that immediately acquires the lock in question. It could be any public method that eventually calls some method that acquires that lock.
    - The code-blobs corresponding to interfaces, and abstract method-calls have to be properly reasoned.
    - The Alternate-Code-blobs have to be taken into account, properly. We needs to allow reporting of n-thread, m-lock, deadlocks.

Steps to proceed along:

1. Prioritising: Approximate procedural analysis of Object-Oriented Java Libraries
    - 1 is relatively strait-forward.
    - 2 must be addressed next.
    - 3-a is the way to go, for now. It causes relatively less work.
    - 5 is the next thing to do.
2. Later: Accurate procedural analysis of Object-Oriented Java Libraries

- 6 can be done to make the procedural analysis accurate.

3. Subsequently: Approximate analysis of Object-Oriented Java Libraries
   - 3-b can be done to begin with.
   - 4 is addressed as a part of that. Check what results are fetched from this.
4. Eventually: Scientific Analysis of OO Java Libraries
   - 7 Tighten the cycle-detection & reporting logic to report plausible cycles.

Chapter 4

Implementation

In this section of our writeup, we shall develop the specific classes that build the CCG for the Java and the C# Language Libraires / Code-bases seperately. In the first section below we develop the Class that builds te CCG for the Java Language Code. and then in the next section, similarly for the C# Code. Both these classes shall, of course, inherit from the `CodeBlob` Class developed above.

4.1 The Java Case

To begin with, we shall specify the usage specification for the program we develop, and its (program development) environment, to some extent. The program usage should be as below:

java lock-conflicts [class-file(s)-in-cwd | a-single-jar-file] ?

The name of the program (Class) is (say) 'lock-conflicts'. When it is invoked without any a jar-file-name argument, it shall report all lock-aquisition-chains emanating from all public methods of all public classes in the named jar. When it is invoked with one (or more) fully-specified class-names, it shall report those subset of lock-aquisition-chains which emanate from the public methods of those classes, and any classes referenced, and their transitive closure. Alternatively the user can specify a set of names of class-files from the *current working directory*.

There seem to be a very large number of options to choose the implementation architecture amongst. After spending nearly one month (of elapsed time) reviewing some of them, we choose the same strategy that we used in Section 2, while 'Investigating the problem-size'. We also re-use all the macros from the section 2.3.1. By doing so, we are able to generate the textually rendered Java-Byte-code, and read it off the `textIn` variable of type `BufferedReader`.

Moreover, we also inherit from the implementation of the `CodeBlob` Class. We are therefore left with merely having to parse through the textual rendering of the class-files, and to identify appropriate points in the input stream when appropriate calls of the functionality exported by the parent class are to be made with the right parameters. We do this as enumerated below:

1. To begin with, we re-use the macros *Read File-name declaration*, and *Read Java Class-name* to do their work. After these macros have been able to do their work, the input stream would have member, and method definitions until the class-close is flagged by a close-brace.

2. Once the body of the class begins, we are interested in the method and its definition, which we need to process. All other fields, and constant definitions we merely ignore. We process the method definitions in two parts: the method-header first, and the method body later. The method-header is processed as under:

   ⟨ Method Header Processor 79 ⟩ ≡

```
static CodeBlob
readMethodHeader (String cName, String line) throws Exception {
    int openP=-1, closeP=-1;
    while (line != null)  {
        if (line.indexOf("}") == 0) return null; // Reached end of Class-definition
        // System.out.println ("head ::== " + line + "::" + line.length());
        openP  = line.indexOf("(");  closeP = line.indexOf(")");
        if ((openP>=0 && closeP>=0 && openP<closeP)  ||  line.indexOf("static {};")>=0)
            break;
        line = textIn.readLine();
    }
    if (line == null)  return null;  // Reached end of Input !! (before end of class ?)

    CodeBlob rv = null;
    String methodName, decoration = "", returnType = null;
    if (line.indexOf("static {};") < 0)  {
        int lastBlankB4OpenP = line.substring(0, openP).lastIndexOf(" ");
        // System.out.println ("Cur Line ::== " + line + "::" + line.length());
        methodName = line.substring(lastBlankB4OpenP+1, openP);
        if (lastBlankB4OpenP >= 0)
            decoration = line.substring(0, lastBlankB4OpenP);

        // Unfortunately, the 1.3.1 API doesnot include the java.lang.String.split() function.
        String parameters = line.substring(openP+1, closeP).trim();
        int    count      = parameters.length() > 0  ?  1  :  0;
        for (int kk = 0;  parameters.length() > kk;  kk++)
             if (parameters.charAt(kk) == ',')   count++;
        String[] params = new String[count];
        if (count == 1)
            params[0] = parameters;
        else if (count > 1) {
            for (int c = 0;  c < count-1;  c++)  {
                int jj = parameters.indexOf(",");
                params[c]  = parameters.substring (0, jj);
                parameters = parameters.substring(jj+1);
                jj = parameters.indexOf(",");
            }
            params[count-1] = parameters;
        }
        // String[] params   = line.substring(openP+1, closeP).split(",");
        // System.out.println ("Params: "+line+" broken into " + params.length + " components.");
        // for (int j = 0;  j < params.length;  j++)
        //     System.out.println (params[j]);


        if (cName.replace('/', '.').equals(methodName))
            methodName = "\"<init>\"";  // Encountered Constructor. ==> returnType is null.
        else  {
            // System.out.println ("cName is " + cName + "; mName is " + methodName);
            int lastBlankB4MethodName = decoration.lastIndexOf (" ");
            if (lastBlankB4MethodName < 0)  {
                returnType = decoration;
                decoration = "";
            }  else  {
                returnType = decoration.substring (lastBlankB4MethodName + 1);
                decoration = decoration.substring (0, lastBlankB4MethodName);
            }
        }
```

```
            rv = newCodeBlob (cName, methodName, getPRTcode(params, returnType),
                     decoration.indexOf("public")>=0, decoration.indexOf("static")>=0,
                     decoration.indexOf("synchronized")>=0, decoration.indexOf("native")>=0);
        }  else  {
            decoration = line.substring (0, line.indexOf("{}"));
            rv = newCodeBlob (cName, getUniqueName(), "()", false, true,
                     decoration.indexOf("synchronized")>=0, false);
        }
        return rv;
    }
    ◇
```

Macro referenced in 83.

3. The parameter-list, and the return-type of a method header are processed by the `getPRTcode` function (called above) as defined below to fetch a type-signature for the method.

⟨ Parameters, and return-type Encoding 80 ⟩ ≡

```
    static String getPRTcode (String[] params, String rt)  {
        String rv = "(";
        for (int i = 0;  i < params.length;  i++)
            rv += encodeType (params[i].trim());
        return rv + ")" + (rt==null ? "V" : encodeType(rt));
    }

    static String encodeType (String type)  {
        String rv = "";
        while (type != null  &&  type.length() != 0  &&  type.endsWith("[]"))  {
            rv += "[";
            type = type.substring (0, type.length()-2);
        }
             if ("byte"   .equals(type))   rv += "B";
        else if ("char"   .equals(type))   rv += "C";
        else if ("double" .equals(type))   rv += "D";
        else if ("float"  .equals(type))   rv += "F";
        else if ("int"    .equals(type))   rv += "I";
        else if ("long"   .equals(type))   rv += "J";
        else if ("short"  .equals(type))   rv += "S";
        else if ("void"   .equals(type))   rv += "V";
        else if ("boolean".equals(type))   rv += "Z";
        else if (type!=null  &&  type.trim().length()>0)  rv += "L" + type.replace ('.', '/') + ";";

        return rv;
    }◇
```

Macro referenced in 83.

4. The definition of the method-behaviour is to be processed. Its presence is flagged by the occurrance of the **Code:** label, on a line by itself, immediately following the method header. The end of the method body is detected by the occurrance of an empty-line.

   Specifically, we need to detect, and processes the ocurrance of three types of tokens: the *invoke* family of op-codes, that correspond to a call to another method, the *monitorenter* token that corresponds to the begining of a *synchronized* block of statements, and the second occurrance of the *monitorexit* token following the coourrance of the matching (opening) *monitorenter* token. The second *monitor-exit* flags the close of the *synchronized* block of statements. The invocations made while within a *synchronized* block of statments have to get registered into the `calledCodeBlobs` member of the newly created null-name node.

⟨ Method Body Processor 81 ⟩ ≡

```
static void processMethodBody (CodeBlob n, String cn) throws Exception {
    boolean firstExitSeen = false;  String line;
    while ((line=textIn.readLine()) != null)  {
        //   System.out.println ("body ::== " + line + "::" + line.length());
             if (line.length()==0)                                     break;
        else if ((line.indexOf("\tinvokevirtual\t")>0) || (line.indexOf("\tinvokespecial\t")>0) ||
                 (line.indexOf("\tinvokestatic\t")>0)) {
            String ms = line.substring (line.indexOf("; //Method ") + "; //Method ".length());
            String mn = ms.substring (0, ms.indexOf(":"));
            ⟨ Replace calls to Array-Clone 82 ⟩
            n.noteCallTo ((mn.indexOf(".")<0 ? (cn.replace('.', '/') + ".") : "") + ms);
        }  // else if (line.indexOf("\tinvokeinterface\t")>0)
           //  n.noteCallTo (line.substring (line.indexOf("; //InterfaceMethod ") +
           //                                             "; //InterfaceMethod ".length()));
        else if (line.indexOf("\tmonitorenter")>0)
            processMethodBody (n.noteCallTo(null), cn);
        else if (line.indexOf("\tmonitorexit")>0  &&  firstExitSeen)   break;
        else if (line.indexOf("\tmonitorexit")>0  &&  !firstExitSeen)  firstExitSeen = true;
        else if (line.indexOf("Exception table:")>=0)  {
            boolean ClassSeen = false, nearEnd = false;
            while((line=textIn.readLine()) != null)  {
                if (nearEnd   &&  line.length()==0)    return;
                nearEnd    = (ClassSeen  &&  line.length()==0)  ||  !ClassSeen;
                ClassSeen  = line.indexOf("Class")>0;
            }
        }
    }
}◇
```

Macro referenced in 83.

5. Correction for calls to the `clone` functionality: After executing the program under development, we detected calls to the natively-implemented `java.lang.Object.clone` functionality that were being threaded through calls to Array-type objects created by the user applications / library compoenents. These were then being reported as undefined functions. For instance: "[Lcom/sun/java/util/jar/pack/Coding;".clone:()I

   We handled this case by replacing such nodes by references to the already created node corresponding to the `java.lang.Object.clone` node, as follows:

⟨ Replace calls to Array-Clone 82 ⟩ ≡

```
if (mn.charAt(0)=='\"'  &&  mn.endsWith("\".clone"))
    ms = "java/lang/Object.clone:()Ljava/lang/Object;";
◇
```

Macro referenced in 81.

6. The above defined methods are integrated into the class definition as follows:

`"JavaBlob.java"` 83 ≡

```
⟨ JavaFile Opener 84 ⟩
public class JavaBlob extends CodeBlob implements Runnable {
    static BufferedReader textIn  = null;
    static String         classesHome = ".";
    static Collection     classesList = null;
    ⟨ Get Class Names From the Jar-file 48 ⟩
    ⟨ JavaP Invoker Thread 52 ⟩
    ⟨ Method Header Processor 79 ⟩
    ⟨ Parameters, and return-type Encoding 80 ⟩
    ⟨ Method Body Processor 81 ⟩
    public JavaBlob () { super (""); }
    public static void main (String[] args) {
        String className="", methodName="", parentName="", line=null;
        try {
            ⟨ Populate classesList based upon usage 51 ⟩
            new Thread(new JavaBlob()).start();
            Thread.currentThread().sleep (2000); // milliseconds
            while (textIn != null)  {
              synchronized (textIn)  {
                while ((line=textIn.readLine()) != null)  {
                  if (line.indexOf("class ")>=0  &&  line.indexOf("{")>=0)  {
                    boolean isAbstract = line.indexOf("abstract ") >= 0;
                    ⟨ Read Java Class-name 61 ⟩
                    inheritanceMap.put (className.replace('.', '/'),
                                        parentName == null ? null : parentName.replace('.', '/'));
                    if (isAbstract)
                        abstractClasses.add (className.replace('.', '/'));
                    while ((line=textIn.readLine()) != null)  {
                        if (line.length() == 0)       continue;
                        if (line.indexOf("}") == 0)   break;
                        CodeBlob m = readMethodHeader(className.replace('.', '/'), line);
                        if (m != null)
                            processMethodBody (m, className);
                        else
                            break;
                    }
                  }
                }
              }
              Thread.currentThread().yield();
            }

            defineReferencedInheritedFunctions ();
                  // Moves nodes from Undefined list into the defined list of nodes.

            SynchCodeReachedFromSynchCode (); // Step 2.1
            dropUnsynchronizedCodeBlobs ();   // Step 2.1.3
            // findCycles (null, null, 5);       // Cycle finding & reporting Step.
            report2LockDeadlockingThreads();
            System.out.println ("LHC: " + lhcs.size()  + ":" + lhcs.toString());
            System.out.println ("PHC: " + t12hc.size() + ":" + t12hc.toString());
        } catch (Exception ex) {
            ex.printStackTrace(System.out);
        }
    }
}◇
```

⟨ JavaFile Opener 84 ⟩ ≡

```
/* CodeBlob.java -- A Node in the Concurrency-Conflict Graph
 *
 * author: Vivek K. Shanbhag, Philips Research, India, Bangalore.
 */

import java.util.*;
import java.io.*;
```

◇

Macro referenced in 66, 83.

Chapter 5

Deadlocks detected

With the help of our first level analysis that we call "Approximate procedural analysis of Object-oriented Libraries", we are able to detect 14 potential deadlocking sceanarios in the JVM, version 1.4.2_08, 17 in version 1.5.0_08, and 19 in version 1.6.0-beta. Of these total of 50 cases, some of which are repeats, we investigated 5 seperate cases. One of them, reported below turned out to be an actual deadlock.

5.1 The Annotations case

One of the potential deadlocks identified by our analysis detected, and reported as below prompted us to develop a simple program that would indeed deadlock the executing JVM instance.

⟨ Annotation Library deadlock as reported by the our analysis 85 ⟩ ≡

```
<2-Cycle java/lang/Class sun/reflect/annotation/AnnotationType.class>
    <Thread-1 Option>
java/lang/Class.initAnnotationsIfNecessary:()V
sun/reflect/annotation/AnnotationParser.parseAnnotations:([BLsun/reflect/ConstantPool;Ljava/lang/Class;)Ljava/
sun/reflect/annotation/AnnotationParser.parseAnnotations2:([BLsun/reflect/ConstantPool;Ljava/lang/Class;)Ljava
sun/reflect/annotation/AnnotationType.getInstance:(Ljava/lang/Class;)Lsun/reflect/annotation/AnnotationType;
    <\Thread-1 Option>

    <Thread-1 Option>
java/lang/Class.initAnnotationsIfNecessary:()V
sun/reflect/annotation/AnnotationType.getInstance:(Ljava/lang/Class;)Lsun/reflect/annotation/AnnotationType;
    <\Thread-1 Option>

    <Thread-1 Option>
java/lang/Class.initAnnotationsIfNecessary:()V
sun/reflect/annotation/AnnotationParser.parseAnnotations:([BLsun/reflect/ConstantPool;Ljava/lang/Class;)Ljava/
sun/reflect/annotation/AnnotationParser.parseAnnotations2:([BLsun/reflect/ConstantPool;Ljava/lang/Class;)Ljava
sun/reflect/annotation/AnnotationParser.parseAnnotation:(Ljava/nio/ByteBuffer;Lsun/reflect/ConstantPool;Ljava/
sun/reflect/annotation/AnnotationType.getInstance:(Ljava/lang/Class;)Lsun/reflect/annotation/AnnotationType;
    <\Thread-1 Option>

    <Thread-2 Option>
sun/reflect/annotation/AnnotationType.getInstance:(Ljava/lang/Class;)Lsun/reflect/annotation/AnnotationType;
sun/reflect/annotation/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>
    ◇
```

Macro never referenced.

5.1.1 The associated Java Application

We pick the annotation code for `Test.java` below, from the sun-site:
http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

```
"Test.java" 86 ≡
    import java.lang.annotation.*;
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public @interface Test {}
    ◇
```

The short multi-threaded program that does deadlock the jvm (1.6.0-beta2-b86, 1.5.0_08-b03) is given below:

```
"Dl.java" 87 ≡
    import java.lang.annotation.Annotation;
    import sun.reflect.annotation.*;
    public class Dl  {
        String className;
        public Dl (String s) {
            className = s;
            new Thread1().start();
            new Thread2().start();
        }
        public static void main(String[] args)  {
            Dl dl = new Dl(args[0]);
        }

        class Thread1 extends Thread  {
            public void run()  {
                try {
                    Annotation[] annArray = Class.forName(className).getAnnotations();
                    for (Annotation a: annArray)  System.out.println (a);
                } catch (Exception e) {
                    System.out.printf ("Thread1 failes: %s %n", e.getCause());
                }
            }
        }

        class Thread2 extends Thread  {
            public void run() {
                try {
                        AnnotationType test = AnnotationType.getInstance (Class.forName(className));
                        System.out.println (test);
                } catch (Exception e) {
                        System.out.printf ("Thread2 failes: %s %n", e.getCause());
                }
            }
        }
    }
    ◇
```

5.1.2 State of the deadlocked JVM

We compile, and run the program as: `javac Test.java Dl.java; java Dl Test`. The executing JVM occassionally deadlocks and can print the below reproduced state whenever prompted.

⟨ Newly detected deadlock through this effort 88 ⟩ ≡

```
Full thread dump Java HotSpot(TM) Client VM (1.5.0_08-b03 mixed mode, sharing):

"DestroyJavaVM" prio=1 tid=0x08acc730 nid=0xc6b waiting on condition [0x00000000..0xbfd8ba60]

"Thread-1" prio=1 tid=0x08b429d8 nid=0xc74 waiting for monitor entry [0xb1fc8000..0xb1fc9040]
        at java.lang.Class.initAnnotationsIfNecessary(Class.java:3028)
        - waiting to lock <0x8cb170e8> (a java.lang.Class)
        at java.lang.Class.getAnnotation(Class.java:2989)
        at sun.reflect.annotation.AnnotationType.<init>(AnnotationType.java:104)
        at sun.reflect.annotation.AnnotationType.getInstance(AnnotationType.java:64)
        - locked <0x8cb16d68> (a java.lang.Class)
        at Dl$Thread2.run(Dl.java:31)

"Thread-0" prio=1 tid=0x08b413d8 nid=0xc73 waiting for monitor entry [0xb2049000..0xb2049fc0]
        at sun.reflect.annotation.AnnotationType.getInstance(AnnotationType.java:61)
        - waiting to lock <0x8cb16d68> (a java.lang.Class)
        at sun.reflect.annotation.AnnotationParser.parseAnnotations2(AnnotationParser.java:72)
        at sun.reflect.annotation.AnnotationParser.parseAnnotations(AnnotationParser.java:52)
        at java.lang.Class.initAnnotationsIfNecessary(Class.java:3031)
        - locked <0x8cb170e8> (a java.lang.Class)
        at java.lang.Class.getAnnotations(Class.java:3011)
        at Dl$Thread1.run(Dl.java:18)

"Low Memory Detector" daemon prio=1 tid=0x08b15158 nid=0xc71 runnable [0x00000000..0x00000000]

"CompilerThread0" daemon prio=1 tid=0x08b13bb0 nid=0xc70 waiting on condition [0x00000000..0xb22e4828]

"Signal Dispatcher" daemon prio=1 tid=0x08b12c80 nid=0xc6f runnable [0x00000000..0x00000000]

"Finalizer" daemon prio=1 tid=0x08b0c288 nid=0xc6e in Object.wait() [0xb25e6000..0xb25e7140]
        at java.lang.Object.wait(Native Method)
        - waiting on <0x88b106c8> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0x88b106c8> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=1 tid=0x08b0b550 nid=0xc6d in Object.wait() [0xb2667000..0xb26680c0]
        at java.lang.Object.wait(Native Method)
        - waiting on <0x88b105d8> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:474)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
        - locked <0x88b105d8> (a java.lang.ref.Reference$Lock)

"VM Thread" prio=1 tid=0x08b089a8 nid=0xc6c runnable

"VM Periodic Task Thread" prio=1 tid=0x08b2ff30 nid=0xc72 waiting on condition


Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x08b0d29c (object 0x8cb170e8, a java.lang.Class),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x08b0d2dc (object 0x8cb16d68, a java.lang.Class),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at java.lang.Class.initAnnotationsIfNecessary(Class.java:3028)
        - waiting to lock <0x8cb170e8> (a java.lang.Class)
        at java.lang.Class.getAnnotation(Class.java:2989)
        at sun.reflect.annotation.AnnotationType.<init>(AnnotationType.java:104)
        at sun.reflect.annotation.AnnotationType.getInstance(AnnotationType.java:64)
        - locked <0x8cb16d68> (a java.lang.Class)
        at Dl$Thread2.run(Dl.java:31)
"Thread-0":
        at sun.reflect.annotation.AnnotationType.getInstance(AnnotationType.java:61)
        - waiting to lock <0x8cb16d68> (a java.lang.Class)
        at sun.reflect.annotation.AnnotationParser.parseAnnotations2(AnnotationParser.java:72)
        at sun.reflect.annotation.AnnotationParser.parseAnnotations(AnnotationParser.java:52)
        at java.lang.Class.initAnnotationsIfNecessary(Class.java:3031)
        - locked <0x8cb170e8> (a java.lang.Class)
        at java.lang.Class.getAnnotations(Class.java:3011)
        at Dl$Thread1.run(Dl.java:18)

Found 1 deadlock.
◇
```
Macro never referenced.

In other instances when the JVM scheduled the execution of the two threads differently, so as to not interfere as above, it produces one of the following two outputs on a successful run, to completion.

⟨ Output-1 from a non-deadlocked run of the program (to completion) 89 ⟩ ≡
```
Annotation Type:
   Member types: {}
   Member defaults: {}
   Retention policy: RUNTIME
   Inherited: false
@java.lang.annotation.Retention(value=RUNTIME)
@java.lang.annotation.Target(value=[METHOD])
◇
```
Macro never referenced.

⟨ Output-2 from a non-deadlocked run of the program (to completion) 90 ⟩ ≡
```

@java.lang.annotation.Retention(value=RUNTIME)
@java.lang.annotation.Target(value=[METHOD])
Annotation Type:
   Member types: {}
   Member defaults: {}
   Retention policy: RUNTIME
   Inherited: false
◇
```
Macro never referenced.

5.2 Empirical Evidence of Correctness / Usefulness

5.3 Bugs detected in java-tools!

Below we list screen dumps of valid uses of the javap fetching surprising results:

⟨ Buggy javap behavior 91 ⟩ ≡
```
[vivek@laptop phd]$ /usr/java/jdk1.5.0_08/bin/javap com.sun.crypto.provider.ai
Compiled from DashoA12275
final class com.sun.crypto.provider.ai extends javax.crypto.SealedObject{
    static final long serialVersionUID;
    com.sun.crypto.provider.ai(javax.crypto.SealedObject);
    java.lang.Object readResolve()       throws java.io.ObjectStreamException;
}

[vivek@laptop phd]$ /usr/java/jdk1.5.0_08/bin/javap com.sun.image.codec.jpeg.ImageFormatException
Compiled from "ImageFormatException.java"
public class com.sun.image.codec.jpeg.ImageFormatException extends java.lang.RuntimeException{
    public com.sun.image.codec.jpeg.ImageFormatException();
    public com.sun.image.codec.jpeg.ImageFormatException(java.lang.String);
}

[vivek@laptop phd]$ /usr/java/jdk1.5.0_08/bin/javap com.sun.crypto.provider.ai com.sun.image.codec.jpeg.ImageF
ERROR:Could not find com.sun.image.codec.jpeg.ImageFormatException
[vivek@laptop phd]$
```
◇

Macro defined by 91, 92.
Macro never referenced.

⟨ Buggy javap behavior 92 ⟩ ≡
```
[vivek@laptop phd]$ /usr/java/j2sdk1.4.2_12/bin/javap sun.text.resources.DateFormatZoneData_th
Compiled from "DateFormatZoneData_th.java"
public final class sun.text.resources.DateFormatZoneData_th extends sun.text.resources.DateFormatZoneData{
    public sun.text.resources.DateFormatZoneData_th();
    public java.lang.Object[][] getContents();
}

[vivek@laptop phd]$ /usr/java/j2sdk1.4.2_12/bin/javap sun.text.resources.DateFormatZoneData_tr
Compiled from "DateFormatZoneData_tr.java"
public final class sun.text.resources.DateFormatZoneData_tr extends sun.text.resources.DateFormatZoneData{
    public sun.text.resources.DateFormatZoneData_tr();
    public java.lang.Object[][] getContents();
}

[vivek@laptop phd]$ /usr/java/j2sdk1.4.2_12/bin/javap sun.text.resources.DateFormatZoneData_tr sun.text.resour
Compiled from "DateFormatZoneData_tr.java"
public final class sun.text.resources.DateFormatZoneData_tr extends sun.text.resources.DateFormatZoneData{
    public sun.text.resources.DateFormatZoneData_tr();
    public java.lang.Object[][] getContents();
}

Compiled from "DateFormatZoneData_th.java"
public final class sun.text.resources.DateFormatZoneData_th extends sun.text.resources.DateFormatZoneData{
    public sun.text.resources.DateFormatZoneData_th();
    public java.lang.Object[][] getContents();
}

[vivek@laptop phd]$ /usr/java/j2sdk1.4.2_12/bin/javap sun.text.resources.DateFormatZoneData_th sun.text.resour
ERROR:Could not find sun.text.resources.DateFormatZoneData_tr
```

```
[vivek@laptop phd]$
```
◇

Macro defined by 91, 92.
Macro never referenced.

Bibliography

[JPL3e] Ken Arnold, James Gosling, David Holmes *"The Java Programming Language, Third Edition"* Sun Microsystems.

[PtJVM] Joshua Engel *"Programming for the Java Virtual Machine"* Addison Wesley.

[Lugrm] Leslie Lamport *"Latex User's Guide and Reference Manual"* Pearson Education Asia.