Deadlock-Detection in Java-Library using Static-Analysis

Vivek K. Shanbhag[1], International Institute of Information Technology – Bangalore.
vivek.shanbhag@{gmail.com, iiitb.ac.in}

June 28, 2011

[1]The author is currently a full-time PhD student at IIIT-Bangalore. This work has been under progress since he was first with Sun Microsystems, and then later while with Philips Research Asia - Bangalore. He gratefully acknowledges their support.

Contents

Chapter 1

report

Abstract

Well-written Java programs that conform to the Language and the J2SE-API Specifications can surprisingly deadlock their hosting JVM. Some of these deadlocks result from the specific manner in which the library implementations (incorrectly) lock their shared objects. Properly fixing them can require corrections in the J2SE-source. We use static-analysis to fetch a list of such potential deadlock scenarios stemming from the library, and use it to drive focused investigation into its source. This can help improve the reliability of the Library-design and implementation. A related reliability-metric can also be designed for objective comparison of update-releases to avoid regression (in maintenance-releases) of large Java libraries. The focus of this investigation is to identify, and thereby, help remove deadlock-possibilities in the Java-Library. An initial prototype implementation of our approach has already helped detect a deadlock in the JDK1.5 Annotation API.

1.1 Introduction

Managed languages like Java support *multi-threading*. In association, they also offer a synchronisation construct, `synchronized`, so that programmers can use both of them together to develop useful multi-threaded programs. Along with the Java Language, is also specified the Java Standard Library, a large collection of APIs (Application Programming Interfaces). Its implementation, J2SE, internally uses both these language features. Specifying additions and enhancements to this Java-Library has come under the purview of (a recently constituted) Java Community Process (JCP), until which time already, a lot of APIs had been designed and implemented that (possibly) did not have formal specifications before their implementations were made publicly available (FCS: First Customer Shipment) in the form of J2SE. Use of the `synchronized` language-feature by the library can be (mutually) in conflict with the use of *multi-threading*, either (both) within J2SE itself, or with the application-program.

While the problem of composing concurrent software from independently developed library components has been studied in the research community[**?**], software-developers still continue to use the same (convenient) language constructs rather than upgrade their programming practice to use, instead, the recently introduced workaround, in the from of the 'Concurrency API'[**?**]. The Posix-compliant *pthreads library* was not known to have any similar problems. Possibly since the same API provided for both the multi-threading, as well as the synchronisation needs, its implementation was able to appropriately address them together. Whereas in the case of Java, these two needs are provided by different communities of programmers: the multi-threading language-level API and its implementation is part of the J2SE-API, while the implementation of the `synchronized` language-construct is hendled by the Virtual-Machine implementation.

Section 10.7 of [**?**] discusses such 'Deadlocks' with an example, and observes[1]: "*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 this in the context of current Java-programming practice, raises interesting questions. For instance,

- *Which* programmer should attempt to avert an application program from reaching its deadlock. Is it the application programmer, *or* is it the Library-API programmer(s). *Or* does it include both. None of these communities of programmers have enough global perspective of the entire system to enable the use of *resource-ordering* at a global level.
- Does the *resource ordering* technique apply to the problem at hand? Is it applicable in the industrial context? Does such a methodology scale to large distributed application-development efforts?

The author believes that a Java-programmer wearing a library-developer's hat must, ideally, annotate the exported API with objective information regarding the implementations' use of the `synchronized` (& multi-threading) constructs. Also that tools /

[1] Emphasis introduced by this author

techniques must be developed to facilitate application-programmers to use the information available in such annotated-APIs along with their knowledge about the application to ascertain its deadlock-freedom. The work reported here is work-in-progress, and a step in this direction.

The major contribution of this paper is a static-analysis approach to detect potential deadlocks that stands in face of the (already very large) size of industrial-strength libraries, and the alarming pace at which some of the heavily-used ones (J2SE, J2EE, etc) continue to grow. In the rest of this section we briefly clarify the terminology, revisit semantics of the synchronisation construct, and use an example case to motivate the problem. Section 2 describes our algorithm, with an illustrative running example. Section 3 presents initial results from our analysis of the Java API, including discussion of a bug we discovered in the Annotation API. The subsequent sections discuss related work, conclusions and future work.

1.1.1 Background and Terminology

The Java Language is specified by the Java Language Specification (JLS)[**?**], and its implementation is specified by the Java Virtual-Machine Specification (JVMS)[**?**]. The language-level keyword / construct, `synchronized`, is described by the JLS, and the JVMS introduces the associated byte-code constructs `monitorenter`, and `monitorexit` that the Java-compiler (`javac`) generates (appropriately) on encountering the use of `synchronized` in the source. The JVMS also specifies the interpretation / execution of these byte-code instructions, by the virtual-machine. The Java standard library is specified (traditionally, through Java Documentation (Javadoc) pages, and of-late more formally through the JCP) and implemented as a part of the J2SE effort, including the (multi-)threading API, `java.lang.Thread`.

J2SE is a free offering from Sun Microsystems that implements all of the above. It is also called the Java Development (tool)Kit, JDK. Simplistically, the JDK contains the various Java-tools, and the JRE (Java Runtime Engine). The JRE contains the VM (Virtual Machine), and a bunch of *jar*-files that together implement the various APIs. The source-code for all of these is together called the J2SE-source.

JDK, versions 1.3.1, 1.4.2, 1.5.0, and 1.6.0 are major releases of the technology, that happen (approximately) 12-18 months apart; these are called trains, and they bundle along new functionality in the form of larger APIs. Along each of the trains, are update releases (1.4.2_12, or 1.5.0_3, for instance), typically once a quarter, that bundle along bug-fixes. The API-specification (to the extent possible) remains frozen, along all update releases on a given train. Effectively the Java Standard API grows with every new major release. Sun Microsystems encourages users to remain current with the update release along whichever train they use, and also try hop onto a more recent train, when possible.

1.1.2 The 'synchronized' keyword

This keyword has declarative semantics, used to lock an object being operated upon. When used to decorate a method-definition, depending upon whether (or not) the

method is `static`, the lock can be either an instance-lock, or a type-lock, (respectively). Alternatively, it can be used as: `synchronized (` *expr* `) {` *statements* `}` to indicate that the enclosed *statements* are to be executed after acquiring a lock on the object fetched from evaluation of *expr*. When used in this manner, it translates into a single `monitorenter` and (at-least one) `monitorexit` byte-codes that enclose the code corresponding to the *statements*. Often, depending upon the structure of *statements*, there can be more than one `monitorexit` instructions that match the same `monitorenter` instruction.

1.1.3 Motivating case

`DeadlockTest.java`[2], below, is a good example of a (otherwise) well-formed multi-threaded Java-program that causes the JVM (version 1.4.2_04) to deadlock. It starts two threads of execution: each thread's `run()` method only enquires into the API. They do not modify the state of any shared object. One thread, (of type `T1`), prints the list of all system-properties, and their currently held values, onto the standard output stream. The other thread, (type `T2`), enquires on the `TimeZone` API, but discards the return value.

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

On execution, the above program creates an object of type `DeadlockTest`, invoking its constructor which creates an object each, of the two types: `T1`, and `T2`. That creates associated objects of type `java.lang.Thread`. Invoking their `start()` method passes control to the `run()` method defined by their classes. Once deadlocked, the JVM reports its thread-state when prompted with the "ctrl+break" key-sequence, as (partially) reproduced below. The complete details about this bug are available at [**?**].

```
Full thread dump Java HotSpot(TM) Client VM
(1.4.2_04-b05 mixed mode):
[...]
Found one Java-level deadlock:
============================
"Thread-1":
```

[2] It is compiled & executed through the command: `javac DeadlockTest.java; java DeadlockTest`

```
  waiting to lock monitor 0x086469cc
    (object 0xab9b0560, a java.util.Properties),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x08646a04
    (object 0xafaefed8, a java.lang.Class),
  which is held by "Thread-1"
[...]
Found 1 deadlock.
```

The above deadlock comprises of two threads: `Thread-0`, and `Thread-1`, and two objects: (each of type), `java.lang.Class` and `java.util.Properties`. Each thread manages to acquire one of the two locks, and blocks, awaiting the other lock; thus realising the deadlock. It is interesting to note that *the code* demonstrating this behaviour, and *the objects* involved, are all from the library. This problem has been fixed by correcting the J2SE implementation. The correction decreases the *lock-strength* of one of the participating code-fragments, thereby removing the possibility of the deadlock. See [**?**] for details of the fix. JVM Version 1.4.2_05, for instance, does not deadlock when executing `DeadlockTest`.

1.1.4 Problem Statement

One can easily write a multi-threaded program that deliberately deadlocks the executing JVM, whereas the program in the example above does not try to modify the state of any object (obviously) shared between its two threads. It does the minimal work necessary to uncover a potential deadlock already present in the Library implementation. Clearly, to fix this problem, the Library needed correction. When faced with such a case, the Javadoc pages at java.sun.com specifying the Java API, unfortunately, do not furnish information regarding the locking behaviour of its implementation. This disables a careful programmer from developing 'provably' deadlock-free Java programs.

The 'Concurrency API'[**?**] exports an alternative for the `synchronized` construct. Using which the J2SE Library programmer can properly synchronise between activity in different co-operating threads and can avoid creating more problems of the above nature. However, an objective empirical study of the standard library implementation shows that library designers, and developers still prefer using the `synchronized` construct rather than the new alternative. Possibly, its declarative aspect is appealing and particularly more convenient as compared to the more imperative programming style necessitated by the Concurrency API. Table 1 lists numbers from a quick count of the Library's usage of multi-threading, and synchronisation language-features. *class/ method counts* list the total number of classes and methods defined in all the *jars* of the JRE. The column *synchronised methods/blocks* list the total number of `synchronized` methods and blocks. The last column lists the number of instances where the `java.lang.Thread` API is invoked, or inherited from (in class definitions). Our inference from the numbers is that the introduction of the Concurrency API has not caused any perceptible shift in programmer preference in its favour, yet.

rt.jar Version	class/method counts	synchronised methods/blocks	java.lang.Thread calls/subclasses
1.4.2_14	9295/77861	2182/1555	750/36
1.5.0_12	12779/114209	3001/2280	1237/41

Table 1.1: Estimation of the problem-size.

1.2 Static-analysis of Objct-Oriented libraries

Conservative Static-Analysis of Object-Oriented libraries to detect all *apparent* sources of deadlock necessarily reports a larger set of instances than actually *feasible*. The practical usefulness of such analysis depends upon the extent of *over-reporting*[**?**] (reporting of *infeasible* deadlocks). Clearly, the fewer false-reports, the better. *Under-reporting* is when some actually *feasible* deadlocks go un-detected by an analysis. Even in such (incomplete, or approximate) analysis it is not the case that all reports are *feasible*: an analysis can therefore do both: include some infeasible deadlocks, while also miss out some feasible ones from its report. Ours, currently, is such an analysis. More specifically, we have taken a phased (or incremental) approach to developing and prototype-implementing our analysis algorithm to static-time predict deadlocks in Java libraries:

- Phase 1: Approximate Procedural-analysis of OO libraries
- Phase 2: (Accurate) Procedural-analysis of OO libraries
- Phase 3: Static-analysis of Object-Oriented libraries

Every successive phase of our implementation has handled more features of the Java Language, and modelled the Library less approximately than before. In this section, below, we list the details of how our analysis has improved over the phases. In the rest of the write-up, later, the version of the analysis that we detail is the one from Phase 3. In its current state (Phase 3), specifically :

Ignoring `native` methods: Our investigation does not look at the C-language implementations of `native` methods of classes defined in the library. In effect, we make a simplifying assumption that native methods neither lock any objects, nor do they call-back into the API.

Data is conservatively abstracted away: Our analysis being static-time, we do not track variable-values. Our analysis is conservative: if there appears an invocation of method T from the byte-code for method C, then it is possible that T is called from within C.

1.2.1 Phase 2: (Accurate) Procedural-analysis of OO libraries

Our previous attempt (Phase 2) was actually a *procedural analysis*: It would discount the polymorphism feature (run-time type based method dispatch) of the Java Language. So also, the abstract methods, and interface methods were not handled with due care.

It started by building a call-graph structure to represent the input-source, wherein every method was represnted as a node of the graph, and every invocation of a concrete-method was represented as an edge between the calling and the called node. Moreover, additionally, every `synchronized` compound-statement was represented as a seperate node by itself. This caused there being two types of nodes in the call-graph. But the edges of the call-graph were still homogenous, and their semantics was still preserved from the phase-1 attempt.

In this phase, we (concurrently with building the call-graph) seperately created the Lock-aquisition-graph which represented ojects (collectively) at the level of types in the system, and ran a cycle-detection algorithm on that to fetch the predictions.

In order to represent statement-block-level usage of the `synchronized` feature, we did a two-pass over the relevent method-bodies, using the exception-dispatch-table to identify matching `monitorexit`, and simulating execution of the method-body using operand-types to fill the slots in the local-variable-array, and operand-stack to discover the type of the lock attempted to be aquired by the `monitorenter` op-codes.

1.2.2 Phase 1: Approximate Procedural-analysis of OO libraries

This first-cut implementation of our analysis was termed *approximate* since it additionally ignored the use of `synchronized` code-blobs within a method. Moreover, in its cycle-detection and reporting step, it restricted the cycle reporting to those that involved only 2 locks, and 2 threads. However, even this simple analysis was able to predict a deadlock in the Annotation API that was successfully reproduced[cite bug-details], and reported[cite apsec paper].

1.2.3 Organisation of our implementation

Our implementation defines three types: `CodeBlob` to model peices-of-code that together compose the entire input being analysed, `OTlock` to model a lock that can be aquired and released (on an *object* or a *type*) using the `synchronized` construct of the Java Language, or the `lock` construct of the C# Language, and `ClassInfo`, objects of which are used to collect useful information regarding the various classes parsed during the analysis. These classes, above, are intended to be source-language-independent. Two other classes: `JavaCode` and `CSharpCode` can be developped to analyse the problem for the J2SE, and the .Net CLR, respectively. It is envisaged that these two language specific classes would inherit the analysis algoithms from the `CodeBlob` class.

Specifically, in this section, we develop the algorithms that come together to compose the implementation of two main types: `CodeBlob`, and `OTlock`, and a utility-class: `ClassInfo`. The Java implementations for these are laid-out as below:

```
"CodeBlob.java" 6≡
    ⟨ JavaFile Opener 8 ⟩
    public abstract class CodeBlob {
        ⟨ Graph Structures comprising the state of the Analysis 10a, ... ⟩
        ⟨ Attributes of the CodeBlob Instance 9, ... ⟩
```

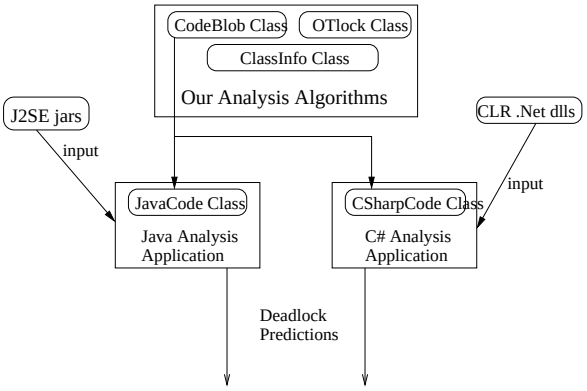

Figure 1.1: Organisation of our implementation.

⟨ *CodeBlob Constructors* 10c, … ⟩
⟨ *CodeBlob Instance Methods* 14a, … ⟩
⟨ *Algorithmic components to transform the graphs, and analyse* 18b, … ⟩
`//` ⟨ *Collect and Report (size) statistics of the analysis* ? ⟩
`}`◇

`"OTlock.java"` 7a≡
⟨ *JavaFile Opener* 8 ⟩
`public class OTlock {`
⟨ *The lock acquisition graph* 15b ⟩
⟨ *Attributes of the lock-object* 15a ⟩
`//` ⟨ *OTlock Constructors* ? ⟩
`//` ⟨ *OTlock Instance Methods* ? ⟩
⟨ *LAG Cycle-detection and reporting* 23, … ⟩
`//` ⟨ *Report statistics of the size of the analysis* ? ⟩
`}`◇

`"ClassInfo.java"` 7b≡
⟨ *JavaFile Opener* 8 ⟩
`public class ClassInfo {`
⟨ *Attributes of the ClassInfo-object* 18a, … ⟩
⟨ *ClassInfo Constructors* 28a ⟩
⟨ *ClassInfo Instance Methods* 28b ⟩
`}`◇

All these classes are designed to firstly represent the aspects that are common to both of the potential input-notations: Java, and C#, and secondly to have an implementation that can be shared between the two different applications necesary to analyse the language-library-sets of the two different input-notations. More specifically, notice that the `CodeBlob` class is an abstract class. The reason for this is that objects that are subject to this analysis have to be instanciated by applications that are aware of the specific syntactical details of the input, (which will be different for Java *vs* C#), whereas, once the entire input has been represented in the form of the call-graph, and related structures, their subsequent transformations, and analysis will be done in a notation-independent manner. There will therefore be some abstract method members of `CodeBlob` that will be provided with implementations in `JavaBlob`, and `CSharpBlob` sub-classes.

⟨ *JavaFile Opener* 8 ⟩ ≡

```
/* file name -- Lock Acquisition Graph Construction, Cycle detection / reporting
 *
 * author: Vivek K. Shanbhag, IIIT-B, Bangalore, India.
 */

package deadlockPrediction;

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

Fragment referenced in 6, 7ab.

1.2.4 The call-graph, its transformations, and the LAG

We start our analysis from a cleanly built J2SE Workspace, and proceed as detailed, below, in this section. We evolve the data-structures, and the implementation of the operations on them alongside the discusion of the algorith,, and use an illustrative running example to further facilitate clarity. As the running Java example that illustrates the function of every step in the algorithm, consider the code below in which C, D, and E represent library classes.

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

```
public class E  {
  public static void m()    { D.n(); };
  public        void y(C c) { c.z(); };
}
```

Step 0: Specifying input

The input to our algorithm is the collection of jar files / .NET dlls (under the `jre / CLR` sub-directory) produced by a successful build of the j2se-workspace (C# CLR). Their successful build implies that the source is well-formed; this way we minimise our work. Our implementation, therefore, need not be robust in the face of ill-formed classes, nor those from bad Java/C#.

Step 1: Single-pass call-graph construction

Iterating over all methods (while also parsing their 'bodies') of all classes in the input, for each method, execute the sub-steps 1.a through 1.g (detailed) below.

Step 1.a: Create a node of the call-graph

Corresponding to every method, whose method-body is either parsed, or whose method-invocation is noticed (while parsing the method-body of some other invoking method), we create a node to represent it. Typically, the fully-qualified name of a method would be an essential attribute of the data-structure that representes it; however, we choose to represent it differently. Since the collection of such nodes that represents the entire input to our analysis is stored in keyed maps, rather than in un-keyed data-collections, we use the two-tuple <fully-qualified-method-name-as-key, node-to-represent-it> as the way to represent/store this information. The data-attributes about the method that we detect, and store as as under:

⟨ *Attributes of the CodeBlob Instance* 9 ⟩ ≡

```
    private String  lockType = null;
        // Stores the type of object that is synchronised upon:
        //  Null implies an 'un-Synchronised' code-blob
        //  A ".class" postfix for static methods after package-name/class-name
        //  TODO: Necessariely maps into the locks hashMap and updates it whenever se
        //  Map-Key: String "package-name/class-name.method-name:prtEncoding"
    public boolean isPublic = false;   // Is this a public method ?
    ◇
```

Fragment defined by 9, 13c, 20a.
Fragment referenced in 6.

The algorithms maintain two hash-Maps, `unexploredCode`, and `exploredCode`; each of which maintain a (1-1) map of method-signatures to corresponding `CodeBlob` nodes. We want to create a node corresponding to every method whose definition is specified in the input, and insert this *fully-specified* node into the `exploredCode`

map with its method-signature as the key. When creating the node, if it is found in the other Map, namely `unexploredCode`, then fetch it from there and populate its attributes before moving it into the `exploredCode` map. Further-more, another hash-map called `sCode` is maintained: the subset of the analysed-code that we are primarily interested to investigate cycles. The connectivity-paths amongst nodes from this subset might visit nodes outside of this subset.

⟨ *Graph Structures comprising the state of the Analysis* 10a ⟩ ≡

```
protected static HashMap   exploredCode = new HashMap(); // Methods-bodies seen
protected static HashMap unexploredCode = new HashMap(); // unseen called methods
          static HashMap          sCode = new HashMap(); // Synchronized methods
```

◇

Fragment defined by 10a, 11c, 13a, 17.
Fragment referenced in 6.

The following constructor helps preserve the program-invarient of the analysis, requiring that there to be at-most one `CodeBlob` instance corresponding to every method in the input-source. For methods that have no behaviour specified in the input to our analysis, we represent them merely by retaining their name in some appropriate list (see Step 1.b). Since the constructor, below, does not have the option of returning an error; its invocation must therefore be conditional. When incorrectly invoked, it detects the inconsistancy being created and asserts failure.

⟨ *Ensure node is not already created* 10b ⟩ ≡

```
assert (  exploredCode.get(@1)==null  &&
        unexploredCode.get(@1)==null);
```

◇

Fragment defined by 10b, 12.
Fragment referenced in 10c, 11a.

⟨ *CodeBlob Constructors* 10c ⟩ ≡

```
protected CodeBlob (String mSignature) {
    ⟨ Ensure node is not already created (10d mSignature ) 10b, … ⟩
    unexploredCode.put (mSignature, this);
};
```

◇

Fragment defined by 10c, 11a.
Fragment referenced in 6.

The constructor above is to be invoked whenever the invocation of a so-far unexplored method is first encountered. Notice that it therefore requires only a single parameter to invoke it, namely the (fully-qualified) name of the method being invoked. Whereas, (as class-definitions are parsed), when method-definitions are encountered for which corresponding nodes have to be created in the call-graph, another method is defined, below, that accepts a lot of information available from the method-header as parameter-values. Notice that the method *checks* that the program-invarient is not violated, before proceeding to create the requested object.

⟨ *CodeBlob Constructors* 11a ⟩ ≡

```
public CodeBlob (String fsClassNm, String mNm,
    String prtEnc, boolean isStatic, boolean isSynch, boolean isPub)  {
    String key  = (fsClassNm +"."+ mNm +":"+ prtEnc).intern();
    ⟨ Ensure node is not already created (11b key ) 10b, ... ⟩
    isPublic = isPub;
    ⟨ Take note of synchronized use 13b ⟩
    exploredCode.put (key, this);
}
◇
```

Fragment defined by 10c, 11a.
Fragment referenced in 6.

Continue to parse the full definition of the method-header-and-body as in steps 1.b through 1.g, below.

Step 1.b: Process `native` & `abstract` methods, and Interfaces

The input may contain a method whose implementation is *native*. While the header information for such methods is available, their implementation (behaviour specification) is empty; We can therefore not record information regarding any invocations that such methods might make. Similarly, (since Java is an object-oriented Language) the library contains definitions of *interfaces*, or *abstract Classes*, containing *abstract Method* members. Interface methods have no implementation. Similarly for *abstract methods*.

Since we do not have any detailed information regarding such methods that do not have implementations specified in our input, we merely remember their names in the appropriate list, defined below:

⟨ *Graph Structures comprising the state of the Analysis* 11c ⟩ ≡

```
public static Set   nativeMethods = new HashSet(); // List  native methods
public static Set abstractClasses = new HashSet(); // List abstract Classes
public static Map abstractMethods = new HashMap(); // Key: abstract mthd nm
public static Map      interfaces = new HashMap(); // Key: Interface name
◇
```

Fragment defined by 10a, 11c, 13a, 17.

Fragment referenced in 6.

What is seen as an invocation of an abstract-method / interface-method (at static-analysis time), translates into (at run-time) transfer of control to some method (with a compatible signature) that provides a concrete implementation. An entry in the `Map abstractMethods`, therefore, stores the name of the abstract-method as the key, and the `Set` of all concrete methods that can pottentially be invoked, instead, as the associated value. Similarly, an entry in the `Map interfaces` stores the name of the `interface` as the key, and the `Set` of all classes that implement it as the associated value.

The interesting aspect about abstract-methods or native-methods is that their *no-implementations-specified*-nature is *not* inferable at the point of their *invocation*. Interface-methods, on the other hand, are invoked by a special byte-code `invoke-interface`, and can, therefore, be recognised as being such. Therefore, our implementation takes the approach: assume (at the point of invocation of method (say) `xx.yy(zz)rr`), that, unless there is *contrary evidence*, its behaviour might have a defination that we shall encounter later on in the analysis. And therefore, at the point of its invocation we create a place-holder for it and include it into our list of `unexploredCode`. *Contrary evidence* comprises of, either that the byte-code used to invoke it is `invokeinterface`, or that the name `xx.yy(zz)rr` already figures in one of the two sets: `nativeMethods`, or `abstractMethods`.

Therefore, as our analysis proceeds, we will encounter occassions when we need to remove tuples from the `unexploredCode`, and move only the name of the method (key of the tuple) into one of the two structures: `nativeMethods`, or `abstractMethods`. Whereas, when faced with the request for creation an instance of type `CodeBlob`, the following peice-of-code asserts that a subset of the *contrary evidence* discussed above is true.

⟨ *Ensure node is not already created* 12 ⟩ ≡

```
assert (!  nativeMethods.contains(@1)  &&
        !abstractMethods.keySet().contains(@1));
```

◇

Fragment defined by 10b, 12.
Fragment referenced in 10c, 11a.

Step 1.c: Process `synchronized` methods

Tag the node if the method is either `synchronized`, or `static synchronized`. Additionally, add all such tagged nodes into the hash-map called `sCode`. This sub-graph of the entire call-graph, corresponds to only the `synchronized` methods.

The information required to accomplish this action is available only at the point of definition of a method. Notice that the `newCodeBlob` method definition above receives

this information through its parameters `isStatic` & `isSynch` for use in appropriately re-defining the value of the `lockType` attribute, and to introduce the newly created <name, node> tupple into the `sCode` hash-map, as detailed below.

⟨ *Graph Structures comprising the state of the Analysis* 13a ⟩ ≡

```
static int     synchronizedCBs = 0;  // count        synchronized statements
static int staticSynchronizeds = 0;  // count static-synchronized methods
```

◇

Fragment defined by 10a, 11c, 13a, 17.
Fragment referenced in 6.

⟨ *Take note of synchronized use* 13b ⟩ ≡

```
if (isSynch)  {
    setLockType (key, fsClassNm + (isStatic ? ".class" : ""));
    sCode.put (key, this);
    if (isStatic)  staticSynchronizeds++;
}
```

◇

Fragment referenced in 11a.

Step 1.d: Populate edges of the call-graph

Parse through the method-body, and for every method-invocation (conditional, or otherwise), insert a pair of directed-edges between the current node (*i.e.* the calling node), and the node corresponding to the called method. If the called method is already known to be a `native` method, then there is not much point representing such an arc in the call-graph, since it will not help any more cycles in the LAG (as we shall see later) than otherwise. If there is no node corresponding to the called method (yet), then create it and insert it into the list of `unexploredCode`, since it is still *under-specified* (its method-body hasn't been parsed, yet).

The implementation uses two maps: `calledCode`, and `callingCode` to represent the two directed edges between nodes of our graph structure. They are both initialised to new empty instances of type `HashMap`. They are updated as detailed in method `noteCallTo`. Below, we optimize by recording a caller-called relation at-most once, and only if the called method is not a native method.

⟨ *Attributes of the CodeBlob Instance* 13c ⟩ ≡

```
HashMap calledCode  = new HashMap(),
        callingCode = new HashMap();
```

◇

Fragment defined by 9, 13c, 20a.
Fragment referenced in 6.

⟨ *CodeBlob Instance Methods* 14a ⟩ ≡

```
public void noteCallTo (String caller,
                                                    String called, CodeBlob ca
    called = called.intern();
    if (calledCode.keySet().contains(called) || // Already recorded.
              nativeMethods.contains(called))   // No point recording this.
        return;
    if (! abstractMethods.keySet().contains(called))
        calledBlob.callingCode.put (caller.intern(), this);
    calledCode.put (called, calledBlob);
}
```

◇

Fragment defined by 14ab, 16.
Fragment referenced in 6.

Step 1.e: Process `synchronized` statements

Recall that the *javac* compiler generates *exactly* one `monitorenter` instruction, and *at-least* one *matching* `moniterexit` instruction(s) corresponding to a single `synchronised` block of statements. Such code, enclosed between a `monitorenter` occurrance, and its *last-matching* `monitorexit` statement is represented as a seperate code-blob in the `sCode` set-of-nodes. For a nested use of the `synchronised` block, there shall be a seperate node to represent the ”nested code”. The way to identify the *last-matching* `monitorexit` statements is dealt in Step 1.f below. These nodes do not figure in the <`exploredCode`, `unexploredCode`> sets (since they are not independently named, they cannot be called into from other methods directly). However, the list `sCode` does store references to such nodes. References to these nodes are also stored in the `calledCode` set of the enclosing code-blob: that is either the `calledCode` attribute of the method whose body was being processed, or of the node correspoding to the enclosing pair of `monitorenter monitorexit` statements. If there are two pairs of `monitorenter monitorexit` statements, one after the other within a method-body, then there will be two seperate nodes to represent them, and there will also be two entries in the `calledCode` set of the enclosing method. Methods invoked within the `monitorenter, monitorexit` pair, will be inserted in the `calledCode` attribute of the corresponding node.

The node corresponding to such code is named as a concatenation of the name of the method-node, a ’#’ symbol, the type-name of the object locked by the monitor, and another ’#’ symbol, followed by a counter-value. The counter value is initialised to 0 for every method-body, and counts up every-time the `monitorenter` bytecode is encountered, within it.

⟨ *CodeBlob Instance Methods* 14b ⟩ ≡

```
public CodeBlob (String p, String c, String lT)  {
    System.out.println ("Creating: " + c); //Debug comment
    setLockType (c, lT);
```

```
        noteCallTo (p, c, this);
        sCode.put (c, this);
        synchronizedCBs++;
    }
    ◇
```

Fragment defined by 14ab, 16.
Fragment referenced in 6.

Step 1.f: Identify the last *matching* `moniterexit`

For methods that have `synchronized` blocks of code, it is necessary to do abstract-interpretation of their byte-code. In doing so, the operand-stack associated with the method-execution is simulated (during static analysis) to contain operand *types* instead of values as would be done during actual execution. This helps to firstly identify the type of the object locked by `monitorenter`, and secondly, it facilitates identification of its various *associated* `monitorexit`s. Towards the close of the byte-code listing corresponding to the behavioral specification of such a method, is the exception-dispatch table. The list of entries in this table are also used to identify the *last-matching* `monitorexit` corresponding to a particular `monitorenter` occurrance.

Step 1.g: Create `otLock` nodes of the LAG

For every non-null value (say *lockName*) assigned to the `lockType` attribute of any `CodeBlob` object, create an `OTlock` object and insert into a global HashMap called `lag`. Recall that *lockName* would be either like "typeName", or like "typeName.Class". Every object of the LAG represents either the lock on the class-object corresponding to the type, or on any instance of that type. In case it represents instance-locks then it alone represents, collectively, all instances of the coressponding type.

Each `otLock` instance (say *from*), being a node of the LAG, has a `lagEdges` attribute that stores the edges of the LAG that start from the *from* node, and connect it to some-other node of the LAG. The value of the `lagEdges` attribute is computed after all the nodes of the LAG are firstly instantiated. Its `lContenders` attribute is the set of names of `CodeBlog` objects that acquire a lock of that type before proceeding with the computation.

⟨ *Attributes of the lock-object* 15a ⟩ ≡

```
    HashMap    lagEdges = new HashMap();
    HashSet    lContenders = new HashSet();
    ◇
```

Fragment referenced in 7a.

⟨ *The lock acquisition graph* 15b ⟩ ≡

```
    static HashMap  locks = new HashMap ();
    ◇
```

Fragment referenced in 7a.

⟨ *CodeBlob Instance Methods* 16 ⟩ ≡

```
/* A public method to set the value for a private attribute
 *  Below is defined 'setLockType' that asserts that the lockType attribute
 *  value is null, and that the value to be assigned to it is non-null.
 *  This assertion enforces the programming discipline that the value
 *  of the private attribute can be set at-most once. The global
 *  initialiser already sets it to null. Only for nodes where it is
 *  discovered later as being 'synchronized', can this method be
 *  invoked, that too only once.
 */
public void setLockType (String myMn, String lock)  {
    assert (lockType == null  &&  lock != null);
    lockType = lock;
    OTlock rv = (OTlock)OTlock.locks.get(lockType);
    if (rv == null)
        OTlock.locks.put(lockType, rv = new OTlock());
    rv.lContenders.add (myMn);
}
public String getLockType () {
    return lockType;
}
public abstract void processMethodBody (String codeBlobName)
                throws java.lang.Exception;
◇
```

Fragment defined by 14ab, 16.
Fragment referenced in 6.

Step 1.h: Note the `public static` *fields* of every class

In step 1, above, we take a procedural view of the library, ignoring polymorphism, and construct a static call-graph. It is easy to see why this computation should terminate: it visits every class in every jar, exactly once, in a finite input. Figure 1 illustrates the call-graph after step 1, corresponding to code-fragment from Step 0. The bold nodes correspond to `synchronized` methods, and the shaded ones indicate `static` methods. A directed edge indicates that the to-node *could be* invoked from execution of the from-node.

Step 2: Inherit parents' methods where necessary

In this step we account for the inheritance feature of Java/C#. Consider an example of Class `B` being a public subclass of Class `A`. Clearly the public interface of `A` is inherited by `B`, and is visible to its users. However, In the Step 1 above, we would not

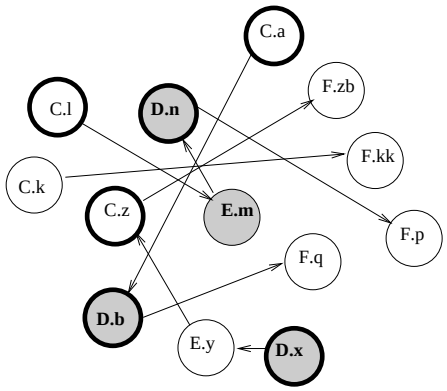

Figure 1.2: The graph constructed after Step-1.

have created any CodeBlob objects for (say) method `aa` defined by `A`, and inherited by `B`, and invoked by some method (say) `C::cc`. So `C::cc` invokes `B::aa`, but we have no definition for it, and whereas we have it only for `A::aa`. After executing Step 1, for this example, `exploredCode` contains nodes `C::cc`, and `A::aa`, and `unexploredCode` contains node `B::aa`. We need to *inherit* the `CodeBlob` object corresponding to `A::aa`, as the definition for method `B::aa`, and move the node for `B::aa` from `unexploredCode` into `exploredCode`. While implementing this inheritatance, care needs to be taken where the method being inherited happens to be either `synchronized` or `static-synchronized`: the 'lockType' of the definition needs to be reset to point to the type of the subclass.

⟨ *Graph Structures comprising the state of the Analysis* 17 ⟩ ≡

```
static protected HashMap classes = new HashMap();
/* The CodeBlob Class has this class-attribute that stores the
 *  ClassInfo object corresponding to every className encountered
 *  during parsing of the input.
 *  mapping of parent-child relationships. Every entry into this
 *  map is a <key, value> pair, where the key is the name of the
 *  sub-class, and the value is the name of the super-class. The
 *  two applications that sub-class from the CodeBlob class must
 *  populate this data, using calls like:
 *  CodeBlob.inheritanceChain.put (subClass.intern(), superClass.intern());
 */
```

◇

Fragment defined by 10a, 11c, 13a, 17.
Fragment referenced in 6.

The information that we store (in the ClassInfo object) corresponding to every class we parse from the input needs to have a field that stores the name of the parent-class, from which it is sub-classed.

⟨ *Attributes of the ClassInfo-object* 18a ⟩ ≡

```
public String parentName = null;
```

◇

Fragment defined by 18a, 26b.
Fragment referenced in 7b.

The first method below, namely `inheritNecessaryParentMethods` loops through all the methods in the `unexploredCode` list, and for each one of them, *walks* up the `inheritanceChain` starting from its containg Class to identify its closest parent from which it can inherit the implementation of that method. For whichever-such undefined method, the defining parent-method can be identified, the task of actual inheriting is delegated to the other method below, namely `inheritParentMethod`.

⟨ *Algorithmic components to transform the graphs, and analyse* 18b ⟩ ≡

```
public static void inheritNecessaryParentMethods ()  {
    int inheritedMethods = 0;
    for (Iterator i = unexploredCode.keySet().iterator();  i.hasNext();  )  {
        String 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  = ((ClassInfo) classes.get(clNm)).parentName;
                    p != null; // p is parent
                    p  = ((ClassInfo) classes.get(p)).parentName)  {
            CodeBlob pc, cc;   // pc: parent code; cc: child code
            if ((pc = (CodeBlob) exploredCode.get (p + rest)) != null)  {
                inheritParentMethod (sign, p+rest, pc);
                inheritedMethods ++;
                break;
            }
        }
    }
    System.out.println ("Inherited " + inheritedMethods + " parent-methods.");
}
```

◇

Fragment defined by 18b, 19, 20b, 22.
Fragment referenced in 6.

The method `inheritNecesaryParentMethods` clearly needs to be a class-method rather than an instance-method since its purpose is to operate upon the class-attribute `unexploredCode`, reducing its size and causing an increase in the connectivity between nodes of the `exploredCode`. The `inheritParentMethod` below, could

well have been an instance method, since it (effectively) populates the `CodeBlob` instance corresponding to the hetherto undefined child-class method, using the parent-class-method-body as a template. However, notice that that too is a `class-static` method, instead. The reason is, (notice) that, in the else part of the *if-then-else* statement, the place-holder `CodeBlob` instance from the `unexploredCode` map corresponding to the inheriting method definition is done-away with, and replaced (instead) by the `CodeBob` instance corresponding to the parents' method. The information regarding invoking-methods from the place-holder instance is used to update the parent-methods `CodeBlob`, as appropriate.

⟨ *Algorithmic components to transform the graphs, and analyse* 19 ⟩ ≡

```
static public void
inheritParentMethod (String cMnm, String pMnm, CodeBlob pM) {
    System.out.println ("Creating " + cMnm + " from " + pMnm);
    CodeBlob cM = (CodeBlob) unexploredCode.remove (cMnm);
    assert (cM != null);
    if (pM.lockType != null)  {
        cM.isPublic  = pM.isPublic;
        cM.setLockType (cMnm, (cMnm.substring(0, cMnm.indexOf('.')) +
            (pM.lockType.endsWith(".class")  ?  ".class"  :  "")).intern());
        cM.calledCode.putAll (pM.calledCode);
        exploredCode.put (cMnm, cM);
        sCode.put (cMnm, cM);
    }  else  {
        for (Iterator i = cM.callingCode.keySet().iterator();  i.hasNext();  )  {
            String caller = (String) i.next();
            if (exploredCode.get(caller) != null)
                ((CodeBlob)exploredCode.get(caller)).calledCode.put (cMnm, pM);
        }
        // pM.callingCode.putAll (cM.callingCode); Unnecessary, isnt it!
        exploredCode.put (cMnm, pM);
    }
}
```

◇

Fragment defined by 18b, 19, 20b, 22.
Fragment referenced in 6.

Step 3: Identify concrete behaviors for abstract methods

Step 3: Discover *reachability* within the `sCode` set

Iterate over nodes of the `sCode` set, populating their `sEdges` attribute (type: `Set`) as follows: insert a directed-edge between two nodes of this set wherever there is a directed-path from one to the other. Additionally, decorate this edge with an hyphenated concatenation of the names of nodes along the path between them. This edge denotes a *direct or eventual* potential call, with possibly other non-`sCode` code-blobs

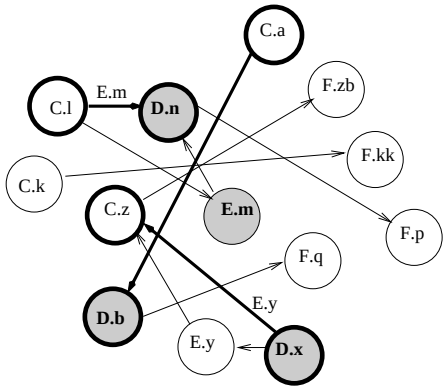

Figure 1.3: The graph after executing Step 3.

in the call-stack, in between the stack-frames corresponding to the caller-method, & the called-method.

An instance attribute `sEdges` of type (`HashSet`) of the `CodeBlob` class is used for this purpose. It is initialised to a null value. When computing this for nodes in the `sCode` set, we instantiate a set object to hold the result of the computation. So also, a type-wide method, and an instance method are specified below that compute the value for this attribute. the computed value for `sEdges` is a set of ascii-lines, one for each edge, and the annotation of the hyphenated-method-names (along the path) is the ascii-text.

Figure 2 illustrates the graph after execution of step 2. The bold edges correspond to the newly-added `sEdges`.

⟨ *Attributes of the CodeBlob Instance* 20a ⟩ ≡

```
HashSet sEdges = null;
```

◇

Fragment defined by 9, 13c, 20a.
Fragment referenced in 6.

⟨ *Algorithmic components to transform the graphs, and analyse* 20b ⟩ ≡

```
public static void sCodeReachedFromSCode ()  {
    System.out.println ("Native method count (definitions not seen): " +
                        nativeMethods.size() + ".\n" + "Non-Synch method count: "
                        (exploredCode.size() - sCode.size()) + ".\n");
    System.out.println ("Synch-method count: \t" +
                        (sCode.size()-synchronizedCBs-staticSynchronizeds) + ".")
    System.out.println ("Static-Synch-method count: \t" + staticSynchronizeds + "
    System.out.println ("Synch-Code-Blobs count: \t" + synchronizedCBs + ".");
    System.out.println ("Total Synch Code count: \t" + sCode.size() + ".");
    for (Iterator i = sCode.keySet().iterator();  i.hasNext();  )  {
        String mNm  = (String) i.next();
```

```
            CodeBlob mB = (CodeBlob) sCode.get(mNm);
            if (mB.calledCode != null)  {
                HashSet workArea = new HashSet();
                workArea.add (mNm);
                mB.augmentReachableSCode (mB.sEdges = new HashSet(), workArea, mNm);
            }
        }
    }

    void augmentReachableSCode (Set rv, Set workArea, String callStack)  {
        if (calledCode != null)
            for (Iterator j = calledCode.keySet().iterator();  j.hasNext();  )  {
                String m   = (String) j.next();
                CodeBlob b = (CodeBlob) calledCode.get(m);
                boolean addedLock = false;
                if (b.lockType != null)  {
                    rv.add (callStack + "-" + m);
                    // addedLock = workArea.add (b.lockType);
                }
                if (workArea.add (m))  {  // an 'else' here gets us edges of the lag
                    b.augmentReachableSCode (rv, workArea, callStack + "-" + m);
                    // workArea.remove (m);
                    // if (addedLock)
                    //    workArea.remove (b.lockType);
                }  // Removing the 'else' above prints paths, from the lag
            }
    }
    ◇
```

Fragment defined by 18b, 19, 20b, 22.
Fragment referenced in 6.

In the two methods defined above, the `workArea` parameter is used to ensure that we do not get into loops that involve re-visiting either the same `CodeBlob` or code that aquires a lock that have already been aquired along the path being explored. This ensures that the computed edge-set is a finite-value.

Step 4: Drop all non-`synchronized` code representation

we delete from the graph, all nodes, (and edges between them), that do not represent `synchronized`, nor `static synchronized` code. Beyond this step in our algorithm, we are interested only in members of the set `sCode`, and `sEdge` amongst them.

Each node in the set (`sCode`) corresponds to the act of first acquiring a lock (or having to block on it), then doing some (useful) work, and finally releasing the lock. The directed-edge `sEdges` represents the *possibility* of a thread trying to acquire *another* lock, while already holding on to the ones acquired before. Of course, the lock

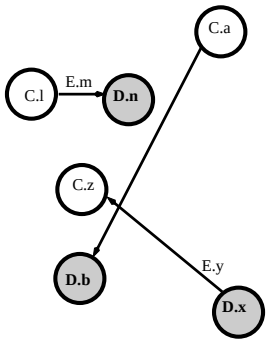

Figure 1.4: The graph after executing Step 4.

could be either on an *object* (instance-lock), or on a *class* (type-lock). Figure 4 illustrates the state of the graph after executing Step 4.

⟨ *Algorithmic components to transform the graphs, and analyse* 22 ⟩ ≡

```
public static void dropUnsynchronizedCodeBlobs ()  {
    exploredCode.clear();
    exploredCode = null;
    System.out.println ("The following are undefined functions (" +unexploredCode
    Object[] undefineds = unexploredCode.keySet().toArray();
    int j = 0;
    for (int i = undefineds.length;  --i >= 0;  )
        if (! ((String)undefineds[i]).startsWith("temp"))  {
            int    iDot = ((String)undefineds[i]).indexOf(".");
            if (iDot < 0)  continue;
            String clNm = ((String)undefineds[i]).substring (0, iDot);
            if (abstractClasses.contains (clNm))  continue;
            System.out.println ("undefined: " + ((String)undefineds[i]));
            j++;
        }
    System.out.println ("Undefined method count: " + j + ".\n");
    unexploredCode.clear();
    unexploredCode = null;
    System.gc();
}
```

◇

Fragment defined by 18b, 19, 20b, 22.
Fragment referenced in 6.

Step 5: Complete constructing the LAG

The nodes from the set `sCode` represent executable code, whereas we want to discover and report cycles on the *OTlock*s that these methods contend for. In this step, we use the

information from the <sCode, sEdges> to discover the edges of the LAG, and record them into `lagEdges`.

Corresponding to a type `tN` in the analyzed library, the `OTlock` named *tN.class* represents the lock associated with the type `tN`, and the `OTlock` named *tN* represents *collectively* all the objects of type `tN`. Each directed-edge in the LAG represents a series of methods, (starting with a method of the class associated with the *calling-code*), one calling into the next, that ends with a method of the class associated with the *called-code*. These two types of nodes that represent instance-locks, and type-locks correspond to locks required for `synchronized` methods, and `static synchronized` methods, respectively. Figure 5 illustrates the LAG, as constructed by Step 5 from the graph in Figure 4.

⟨ *LAG Cycle-detection and reporting* 23 ⟩ ≡

```
static public void computeLAGedges () {
    for (Iterator i = locks.keySet().iterator();  i.hasNext();  )  {
        String lN  = (String) i.next();                    // lock-Name
        if (lN.charAt(0) == '[')  continue;
        Set    lCS = ((OTlock)locks.get(lN)).lContenders;  // lock-Contender-Set
        for (Iterator j = lCS.iterator();  j.hasNext();  )  {
            String lCN   = (String) j.next();   // lock-Contender-Name
            Set    eSlCN = ((CodeBlob)CodeBlob.sCode.get(lCN)).sEdges;
                                                // edges-Starting from lock-Cont
            for (Iterator k = eSlCN.iterator();  k.hasNext();  )  {
                String eAnnot = (String) k.next();
                int firstHyphen = eAnnot.indexOf('-'), lastHyphen = eAnnot.lastIn
                if (firstHyphen < 0  ||  !lCN.equals (eAnnot.substring(0, firstHy
                    ! ((CodeBlob)CodeBlob.sCode.get(lCN)).getLockType().equals (l
                   System.out.println ("Assertion failure would have occurrred in
                   //   asserting that the first component is lCN and its lockTyp

                if (lastHyphen >= 0)  {
                    String tN = eAnnot.substring(lastHyphen+1);  //last method in
                    String tLockType   = ((CodeBlob)CodeBlob.sCode.get(tN)).getLo
                    assert (tLockType != null);
                    OTlock  l = (OTlock) OTlock.locks.get(lN);
                    HashSet tempM = (HashSet)((HashMap)l.lagEdges).get(tLockType)
                    // Create an entry for the tLockType into lN.lagEdges, if not
                    //  the value for this new entry is an object of type HashSet
                    if (tempM == null)
                        l.lagEdges.put (tLockType, tempM=new HashSet());
                    tempM.add (eAnnot);  // Into this hashSet insert the edgeAnno
                }
            }
        }
    }
}
```

◇

Fragment defined by 23, 24, 27f.
Fragment referenced in 7a.

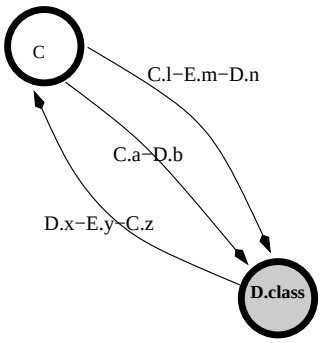

Figure 1.5: The LAG after executing Step 5.

Step 6: Finally, discover and report cycles in the LAG

The method below detects, and reports cycles in the `lag` of length 2..*n* where *n* is its only invocation argument-value.

⟨ *LAG Cycle-detection and reporting* 24 ⟩ ≡

```
static public void reportLAGcycles (int maxLen) {
    assert (maxLen >= 2);
    String  [] cycle      = new String  [maxLen]; // cycle[0] == cycle[n]  ==>  p
    Iterator[] populator  = new Iterator[maxLen]; // iterative method, not recurs
    for (int len = 2;  len <= maxLen;  len++)  {  // start looking for len-cycles
        populator[0] = locks.keySet().iterator(); // initialise the iterator to f
        boolean doSelect = true;
        for (int j = 0;  j < len;  j++)  {
            if (doSelect  &&  j==0  &&  !populator[j].hasNext())
                break;
            doSelect = true;
            while (populator[j].hasNext()  &&  doSelect)  { // selects a 'legal'
                cycle[j] = (String) populator[j].next();
                doSelect = false;
                for (int k = 0;  k < j;  k++)     // If the just populated lock i
                    doSelect |= (cycle[j] == cycle[k]);    //  the cycle, select
                doSelect |= (j > 0)  &&          // Not a legal selection, sorry,
                            !((OTlock)locks.get(cycle[j-1])).lagEdges.keySet().co
                // bad idea!! doSelect |= (j > 0)  &&  (cycle[j].compareTo(cycle[
                // ordering to avoid multiple reporting!
                doSelect |= (j == len-1)  &&     // Not a cycle, too bad, select
                            !((OTlock)locks.get(cycle[j])).lagEdges.keySet().cont
            }
            if (doSelect) {
```

```
                    if (j>0 && !populator[j].hasNext())  { // No legal next node; Bac
                        j--; j--;      // Equivalent to the recursive backtrack; '-2'
                    }
                } else {
                    if (j < len-1)     // If lock-j lock isnt the last lock
                        populator[j+1] = locks.keySet().iterator(); //  then initiali
                    if (j == len-1)
                        if (((OTlock)locks.get(cycle[j])).lagEdges.keySet().contains(
                            printCycle (len, cycle);              // having selected
                            for (int k = j;  k>=0;  k--)
                                if (populator[k].hasNext())  {
                                    j = k-1; // '-1' precorrects for j++; having prin
                                    break;
                                }
                        }
                }
            }
        }
    }

    static public void reportLockUsage ()  {
        System.out.println ("Number of locks used in the code-base: " + locks.size())
        int clam = 0, clab = 0; // cumulative lock acquring methods/blocks counters
        for (Iterator i = locks.keySet().iterator();  i.hasNext();  )  {
            int lam = 0, lab = 0; // lock acquring methods/blocks counters
            String k = (String) i.next();
            for (Iterator j = ((OTlock)locks.get(k)).lContenders.iterator();  j.hasNe
                if (((String)j.next()).indexOf('#') < 0)  lam++;
                else                                      lab++;
            System.out.println (k + ": " + lam + ":" + lab);
            clam += lam;  clab += lab;
        }
        System.out.println ("Cumulative clam = " + clam + ", clab = " + clab + ", Tot
    }
    ◇
```

Fragment defined by 23, 24, 27f.
Fragment referenced in 7a.

For our example, the algorithm would print an XML-format report as reproduced below. The report renders the information in the LAG into text. Notice the tags: `Thread-1`, and `Thread-2`. They indicate that if there were to be two threads in an application program, one of which were to realise any of the stack-traces under the 'Thread-1' (options), and the other were to realise any of those under 'Thread-2' (options) *concurrently*, then the said deadlock *could* occur. In this example, the lock `C` is an instance-lock, whereas the lock `D.class` is a type lock. We term such an XML-output as a single *report*, irrespective of how-many thread-stack options it contains.

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

The `printCycle` method below tries to avoid reporting duplicates. For this purpose, it maintains a `set` of `alreadyPrintedCycles`, and attempts to add every input cycle before proceeding to print it; if the operation of adding into the `set` succeeds, then the cycle has not be reported before, and is therefore printed, else it is not re-printed. `alreadyPrintedCycles` is maintained in the form of a `set` of integer values, each corresponding to the sum of the hash-codes of each of the locks involved in a given cycle.

⟨ *Store cycle-signature and check to avoid repeats* 26a ⟩ ≡

```
int cumulativeHashValue = 0;
for (int i = 0;  i < len;  i++) {
    lockNameList += (" " + cycle[i]);
    cumulativeHashValue += cycle[i].hashCode();
}
if (! alreadyPrintedCycles.add(cumulativeHashValue))
    return;
```

◇

Fragment referenced in 27f.

In the cycle printed by the method, we would also like to print information related to the containers of instances of types whose instance-locks are part of the lag. This helps the act of synthesizing programs to deadlock the JVM as predicted by the cycle, given the source of the library being analysed. The ClassInfo object corresponding to a particular Class `K`, for instance, will collect into its attribute `containerInstances`, all descriptors `name:type`, where the class `type` contains an (non-static) instance attribute `name` of type `K`. Another attribute `containerClasses` collects all descriptors `sName:type`, where the class `type` contains a `static` (class) attribute `sName` of type `K`.

⟨ *Attributes of the ClassInfo-object* 26b ⟩ ≡

```
public Set containerInstances = new HashSet();
public Set containerClasses  = new HashSet();
```

◇

Fragment defined by 18a, 26b.
Fragment referenced in 7b.

⟨ *Print list of Containers of classes associated with instance-locks* 26c ⟩ ≡

```
for (int i = 0;  i < len;  i++)  {
    if (cycle[i].endsWith(".class"))
        break;
    ⟨ Print list of Containers (27a Instance,27b Instances ) 27e ⟩
    ⟨ Print list of Containers (27c Types-as,27d Classes ) 27e ⟩
}
◇
```

Fragment referenced in 27f.

⟨ *Print list of Containers* 27e ⟩ ≡

```
System.out.println ("<@1-Containers-of--" +cycle[i]+ ">");
for (Iterator j = ((ClassInfo)CodeBlob.classes.get(cycle[i])).
                  container@2.iterator();  j.hasNext();  )
    System.out.println (((String)j.next()));
System.out.println ("</@1-Containers-of--" +cycle[i]+ ">");
◇
```

Fragment referenced in 26c.

⟨ *LAG Cycle-detection and reporting* 27f ⟩ ≡

```
static private Set alreadyPrintedCycles = new HashSet();
static public void printCycle (int len, String[] cycle) {
    String lockNameList = "";
    ⟨ Store cycle-signature and check to avoid repeats 26a ⟩
    System.out.println ("<Cycle-" +len+ "locks=\""+lockNameList +"\">");
    ⟨ Print list of Containers of classes associated with instance-locks 26c ⟩
    for (int i = 0;  i < len;  i++)  {
        for (Iterator j = ((HashSet)((deadlockPrediction.OTlock)locks.
                                     get(cycle[i])
                                    ).lagEdges.get(cycle[(i+1)%len])
                           ).iterator();  j.hasNext();  ) {
            System.out.println ("<Thread-" + (i+1) + ">");
            System.out.println (((String)j.next()).replace('-', '\n').
                                replaceAll ("\"<init>\"", "\"init\""));
            System.out.println ("</Thread-" + (i+1) + ">");
        }
    }
    System.out.println ("</Cycle-" + len + ">");
}
◇
```

Fragment defined by 23, 24, 27f.
Fragment referenced in 7a.

The supporting infrastructure in the form of the ClassInfo type is easily completed as detailed below. When parsing the text rendering of the input jar-files or dll-files, the name of the super-class is available in the class-header information. Similarly, the names and the types of the fields (class / instance attributes) of the class (being parsed) is also available as part of class-header information. However, (in the case of Java (as input)) the information that a particular field (attribute) is `static` is available only when (and if) an `putstatic` or a `getstatic` byte-code is encountered when parsing any of the method-bodies (whose fields (or attributes) have already been noted when parsing the header information for the class). Therefore, we have the following call-interface, where the `JavaCode` class implementation firstly intimates the names-&-types of all fields as read from the containing-class header-information, using `setEnclosingInstance` defined below. And then, as and when any `put/get-static` is encountered for some specific field-reference, it is *upgraded* to class-static using the `setEnclosingClass` method below.

⟨ *ClassInfo Constructors* 28a ⟩ ≡

```
public ClassInfo (String pNm) {
    parentName = pNm;
};
```

◇

Fragment referenced in 7b.

⟨ *ClassInfo Instance Methods* 28b ⟩ ≡

```
public void setEnclosingInstance (String desc) {
    containerInstances.add (desc);
};

public void setEnclosingClass (String desc) {
    if (containerInstances.remove (desc))
        containerClasses.add (desc);
};
```

◇

Fragment referenced in 7b.

1.3 Results from analysing J2SE

Using the above algorithm, we analysed the JRE for the current update-release along 2 different trains of the JDK: 1.4.2_08, and 1.5.0_08. Table 2 gives details of the size of the analysis. We detect 14 potential deadlock reports in version 1.4.2_08, 17 in version 1.5.0_08, and 19 in version 1.6.0-beta. Of these (total) 50 cases, (some of which are repeats), we investigated 5 separate cases. One of them, discussed later is an actual deadlock.

JRE	Jars parsed(MB *iff* > 1)	Method counts	cycles
1.4.2_08	rt(27), charsets(6), jce, jsse, sunrsasign, localedata, ldapsec,...	Total: 86677 synch: 2723 native: 1343	14
1.5.0_08	rt(40), charsets(9), jce, jsse, sunpkcs11, sunjce_provider,dnsns,...	Total: 126921 synch: 3759 native: 1729	17

Table 1.2: The size of our analysis.

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

Every report produced from our analysis identifies specifically two *sets* of locks; any one lock from *each set* can be acquired using library-code, in a possibly (mutually) deadlock-ing manner. For the cycle reported in Figure 4, one set of locks is the set of all objects of type `o`. The other set of locks is the *class* object corresponding to the type `p`. The former set contains a potentially large set of objects, whereas the latter corresponds to a singleton set. Favourable conditions necessary for a program execution to reach such a deadlock include:

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

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

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

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

Figure 1.6: A sample deadlock possibility report fetched from JRE analysis.

- If both the code (on top-of-stack), and objects involved in the deadlock are provided by the library, then, unless the actual library-API invoked by the user-code is intended to modify the state of the object it acts upon, since there is no obvious need for the API-implementation to lock its objects, it is possibly J2SE that needs correction.

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

1.3.1 The Annotations case

A potential deadlock identified by our analysis, and reported as in Figure 5, prompted us to develop a program: (`Dl.java`) that realises it. For brevity we reproduce only the relevant thread stacks from the report. Notice that among the two objects involved in the deadlock, one would be of type `java.lang.Class`, and the other would be the Class-object associated with the type `sun.reflect.Annotation.AnnotationType`.

As per the report, the two application threads could deadlock if one of them were to call `Class.initAnnotationsIfNecessary()`, and the other call `AnnotationType.getInstance()`, concurrently. Inspecting `java.lang.Class.java`, reveals that its `initAnnotationsIfNecessary()` method is `private`, which applications cannot directly invoke. However, some of its `public` methods, (for instance: `getAnnotations()`) invokes the `private` method *unconditionally*. Notice that, therefore, `Thread1` of `Dl.java`, invokes this method. `Thread2` invokes `AnnotationType.getInstance()` as required by the report. The parameter to these two methods is of type `java.lang.Class`, corresponding to `Test.java`. Notice that `Test.java` needs to be an annotation-type, so that it can

be a legitimate argument for `AnnotationType.getInstance()`. `Test.java` is from a sun-site: [**?**]. Aspects of the design / implementation of `Dl.java` are inspired from the motivating example. When invoked as in: `javac Test.java Dl.java; java Dl Test`, the JVMs version 1.6.0-beta2-b86, and version 1.5.0_08-b03 do get deadlocked. This is filed as a bug with 'Java sustaining' at Sun Microsystems[**?**].

"Dl.java" ≡

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

"Test.java" ≡

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

1.4 Related Work

This, and similar problems have been studied, with varying degrees of success. Amy Williams[**?**][**?**] studies precisely this problem, and reports a few deadlocks in some of the libraries they investigate, including the JDK. They describe a more sophisticated machinery than ours, which requires larger state for investigation. Consequently their approach is unable to investigate the entire JDK, and when employing approximations similar to ours they are able to cut down the memory requirement to available space. Never-the-less, they report investigating far fewer classes than are available in the distribution. Von Praun [**?**] studies *Synchronisation defects* including deadlocks in entire applications, rather than in libraries as we propose above. He also uses more elaborate modelling techniques than ours, but reports results from investigating smaller codebases than J2SE. In comparison, since our primary aim is to investigate the entire Java Library (which is quite large, and growing quite fast), we have chosen to model its details at a coarser granularity than both Amy, and Von Praun. This helps us to create a

manageable sized abstract *lock-order-graph* that represents the entire JRE-Library, and then refine it to fetch the cycle-reports.

Both these analysis use source-code as input. Amy analyses Library-code using a flow-sensitive, context-sensitive approach, whereas Von Praun investigates entire applications using a context-sensitive approach. In both their algorithms, the analysis is performed at the level of object-instances, whereas we restrict our modelling of the code-base to its underlying type-system. We thereby contain the size of the investigation well within the limits of present day desk-top computing-abilities. Moreover, our reporting format helps us co-locate all necessary information to facilitate developing a deadlocking-program. Also, both their analysis additionally addresses the wait-notify class of deadlocks, that we do not investigate.

Bandera [**?**] extracts a model from the Java-source for entire applications that is used to drive a model-checking approach to program-verification. This may not apply to check libraries.

1.5 Conclusions

We have developed a prototype-program to automatically detect potential deadlock scenarios that are available in the J2SE-APIs. Our simple approximate procedural analysis of Object-oriented libraries, has helped us detect one new bug in a recent and current release of the API implementation. We foresee two important benefits of this approach/tool:

- Address legacy problems:- the cycles identified for recent releases can be used as a starting point to drive focused investigation to improve the reliability of the Platform.
- Its incorporation into regression testing: using this tool in the JDK release process, with appropriate definitions for related metrics, can identify any potential regression along individual trains, in update-releases.

1.6 Future Work

To do a more accurate procedural analysis we must be able to reason about `synchronized` code-fragments, and identify the types associated with their locks. Secondly, we need to handle virtual functions of class-hierarchies. A more scientific and object-oriented analysis involves handling calls to un-implemented methods of abstract classes, or interface-methods. While reporting, we currently limit to 2-cycle, 2-thread deadlocks, whereas in the general case, *n*-cycle, *m*-thread deadlock possibilities are of interest. Simple heuristics that avoid reporting of potential thread-states which include exception-paths can substantially reduce the number of false-negatives reported.

All of the above add up to detecting, and correcting the deadlock possibilities in the libraries. It can be valuable to use all this information to avoid the occurrence of deadlocks when executing code. This would involve the implementation of a deadlock-avoidance algorithm as a part of the executing JVM.

1.7 Acknowledgements

I would like to gratefully acknowledge the guidance from my adviser, Prof. K. V. Dinesha, his patience, and the many useful discussions that helped lead up to the current state of investigation. Valuable comments from Srihari Sukumaran, helped improve the readability of this write-up. Continued help and encouragement from Poonam, and Thilak has been very useful.

1.8 Analysing the J2SE Library

We begin with, developing the usage specification for the program, and exploring its (program development) environment, to some extent. The next thing to do is to evolve the high-level organisation of the program-fragments before geting into the details. The program usage should be as below:

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

We call the program (Class) 'lag-conflicts': Above, the analysis can be requested either for a set of (named) class-files (in the cwd), 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 the default.

Recollect from the discussion under Step 0 of the previous section, that our implementation strategy is to use byte-code as input rather than the Java-source. As a part of the JDK distribution, there is a utility called *javap* that renders byte-code into text. We can use this utility to facilitate our implemenation. However, *javap* being a seperate program, we can design our use of it by starting a concurrent avtivity to disassemble the Java-byte-code into text, and read this text-rendering into the program and process it. Note te following design considerations:

- The 'javap' utility 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. Its implementation searches the 'classpath' for class-files named on the command line, and renders them into text on the standrd output. Since the length of the command-line is an installation specific charachteristic, we would like not to depend upon its value being set to our convinience.

- Also, because of the above mentioned classpath-spec.requirement, we chose 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 used with a mix of the class and any jar file as parameters.

- In this 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 as its invocation parameter), and the second thread to convert the byte-code to text (using *javap*), and then the third (main) thread to read the text generated by the second thread and process it.

- Appropriate 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 (say, 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).

Our implementation of the `JavaBlob` class is organised as follows: JavaFile Opener $JavaBlob$

"JavaBlob.java" 34a≡

```
import java.util.*;
import java.util.regex.*;
import java.io.*;
import deadlockPrediction.*;
public class JavaBlob extends deadlockPrediction.CodeBlob {
    ⟨ JavaBlob Constructors 35a ⟩
    ⟨ Globally accessable state for sharing data amongst threads 35b ⟩
    ⟨ Methods to process invocation & invoke javap to create text-rendering 36a, … ⟩
    ⟨ Methods to parse text-rendering and create initial call-graph structures 41a ⟩
    ⟨ Main method to execute algorithms as per the steps, to print cycles, if any 34b ⟩
}
```

◇

We would like the highest level main method to invoke methods in the order of the Steps listed in the previous section that discussed the algorithm to discover the cycles and report them.

⟨ Main method to execute algorithms as per the steps, to print cycles, if any 34b ⟩ ≡

```
public static void main (String[] args) {
    try {
        readInputNdCreateCallGraph (args);
        inheritNecessaryParentMethods();  // Move nodes from Undefined list into
        deadlockPrediction.OTlock.reportLockUsage();
        sCodeReachedFromSCode ();         // Step 2.1
        dropUnsynchronizedCodeBlobs ();   // Step 2.1.3
        deadlockPrediction.OTlock.computeLAGedges ();        // Cycle finding Ste
        deadlockPrediction.OTlock.reportLAGcycles (7);       // Cycle reporting
    } catch (Exception ex) {
        ex.printStackTrace(System.out);
    }
}
```

◇

Fragment referenced in 34a.

We define one constructor for JavaBlob objects corresponding to every constructor defined by the base class: CodeBlob. The JavaBlob constructor merely invokes the corresponding base-class constructor.

⟨ *JavaBlob Constructors* 35a ⟩ ≡

```
private JavaBlob (String mSignature) {
    super (mSignature);
};

public static String newJavaBlob (String fsClassNm, String mNm, String prtEnc,
    boolean isStat, boolean isSynch, boolean isPub, boolean isNat, boolean isAbst
    String key = (fsClassNm +"."+ mNm +":"+ prtEnc).intern();
    if (unexploredCode.get (key) == null  &&  exploredCode.get (key) == null)
        new JavaBlob (key);
    return newCodeBlob (fsClassNm, mNm, prtEnc, isStat, isSynch, isPub, isNat, is
}

public void noteCallTo (String caller, String called)  {
    if (unexploredCode.get (called) == null  &&  exploredCode.get (called) == nul
        new JavaBlob (called.intern());
    super.noteCallTo (caller, called);
}

public JavaBlob newSJavaBlob (String p, String c, String lT)  {
    new JavaBlob (c.intern());
    return ((JavaBlob) newSCodeBlob (p, c, lT));
}
◇
```

Fragment referenced in 34a.

1.8.1 Disassemble Binaries into Text

The invocation arguments are able to fetch either a list of class-names or a list of jars, whereas the *javap* command requires class-names to fetch the text-rendering of their byte-code. Below we develop the code-fragments to infer the list of class-names intended to be used as input, and the value of the classpath parameter, both of which are needed to invoke the *javap* command.

⟨ *Globally accessable state for sharing data amongst threads* 35b ⟩ ≡

```
static BufferedReader textIn      = null;
static String         classesHome = null;
static Collection     classesList = null;
◇
```

Fragment referenced in 34a.

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.

⟨ *Methods to process invocation & invoke javap to create text-rendering* 36a ⟩ ≡

```
static public void classListGivenUsage(String[] args)  {
    if (args.length==0)
        classesList = classNamesFromJar (null);
    else if (args[args.length-1].indexOf(".class")<0  &&
             args[args.length-1].indexOf(".jar")>=0) {
        classesList = classNamesFromJar (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]);
        classesHome = ".";
    }
}
```

◇

Fragment defined by 36ab, 38, 39.
Fragment referenced in 34a.

We use the output of the ’jar -tvf ...’ command, piped into a ’grep’ for entries ending with a ”.class” to fetch the class-names as required above. This is implemented in the following function.

⟨ *Methods to process invocation & invoke javap to create text-rendering* 36b ⟩ ≡

```
static public Collection classNamesFromJar (String jar)  {
    boolean onlyOne = jar != null;
    try {
        if (! onlyOne)  {
            String version = System.getProperty ("java.version"),
                   home    = System.getProperty ("java.home");
            jar = home + "/lib/rt.jar "   + home + "/lib/jce.jar " +
                  home + "/lib/jsse.jar " + home + "/lib/charsets.jar";
            if (version.startsWith ("1.5"))
                jar += ’ ’ + home + "/lib/ext/sunjce_provider.jar " + home+"/lib/
                           home + "/lib/ext/sunpkcs11.jar " + home + "/lib/ext/
            else if (version.startsWith ("1.4"))
                jar += ’ ’ + home + "/lib/sunrsasign.jar " + home + "/lib/ext/loc
                           home + "/lib/ext/sunjce_provider.jar " + home+"/lib/
                           home + "/lib/ext/ldapsec.jar";
        }
```

```
            Process child = Runtime.getRuntime().exec (
                (onlyOne  ?  "./grepNDcut "  :  "./grepNDcutInLoop ") + jar);
            BufferedReader in   = new BufferedReader (new InputStreamReader (child.ge

            TreeSet retval = new TreeSet();
            for (String line = in.readLine();  line != null;  line = in.readLine())
                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;
    };
    ◇
```

Fragment defined by 36ab, 38, 39.
Fragment referenced in 34a.

The above implementation invokes a shell script that is defined below. It implements the functionality 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" 37a≡

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

"grepNDcutInLoop" 37b≡

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

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 in 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.

⟨ *Methods to process invocation & invoke javap to create text-rendering* 38 ⟩ ≡

```
static final int  NoOfClassesPerJavapInvocation = 20;
public static void runJavapInvoker()  {
    Runnable invoker = new Runnable() {
        public void run() {
            String  subList;       InputStream textOut = null;
            Process child = null;  InputStreamReader r = null;
            try {Iterator i = classesList.iterator();
                do {subList = "";
                    for (int j = NoOfClassesPerJavapInvocation;  --j>=0 && i.hasN
                        String t = (String) i.next();
                        subList = subList + " " + t;
                        if (breakpoint (t))  break;
                    }
                    String delete2be = System.getProperty ("java.home") +
                           "/../bin/javap -private -c -verbose -J-Xmx98m " +
                           (classesHome!=null ? ("-classpath "+classesHome) : "")
                    System.out.println (delete2be.replace (' ', '\n'));
                    child = Runtime.getRuntime().exec (System.getProperty ("java.
                                                "/../bin/javap -private -c -verbos
                        (classesHome!=null ? ("-classpath " + classesHome) : "")
                    r = new InputStreamReader (child.getInputStream());
                    if (textIn==null)              textIn  = new BufferedReader (
                    else  synchronized (textIn) {  textIn  = new BufferedReader (

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

```
}
```
◇

Fragment defined by 36ab, 38, 39.
Fragment referenced in 34a.

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 used above is to protect our analysis from the implications of this bug. It is defined above.

⟨ *Methods to process invocation & invoke javap to create text-rendering* 39 ⟩ ≡

```
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.tex
                fqcn.startsWith ("sun.net.spi") || fqcn.startsWith("sun.net.www")
                fqcn.equals ("com.sun.security.sasl.util.SaslImpl");
     return true;
}
```
◇

Fragment defined by 36ab, 38, 39.
Fragment referenced in 34a.

1.8.2 Parse Text-rendering to create the call-graph

Recollect that `JavaBlob` inherits from the the `CodeBlob` Class. Its implementation therefore merely parses through the text- rendering of the class-files, and at appropriate points in the input stream it calls of the functionality exported by the parent class with the right arguments. A quick look at a sample Java-source, and the corresponding text-rendering from its compiled class-file can easily motivate our design of the organisation for this part of the implementation. The implementation is further detailed below:

Sample Java, and its corresponding text-rendering: We shall briefly look at a short Java-class and the disassembly of its corresponding class-file. We use the 'javac', and 'javap' tools available as a part of the J2SE distribution, for compiling Java, and disassembly of the so generated class-file, respectively.

⟨ *A Short Java Program* 40a ⟩ ≡

```
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);
    }
  }
}◇
```

Fragment 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 plain-text. Below we reproduce the same so as to give the reader a feel for the format of the text that we parse to do our processing in the rest of this write-up.

⟨ *The text-rendering of MLtest.class* 40b ⟩ ≡

```
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/
   8:   return

}◇
```

Fragment never referenced.

The input stream `textIn` has class-name, open-brace, member(s), and method definitions until the class-close is flagged by a close-brace on line all by itself. Once the body of the class begins, we are interested in the method-header and its body-definition, which we need to process. All other fields, and constant definitions we can ignore. We first process the class-declaration. This is followed by method definitions in two parts: the method-header first, and the method body later.

The code for this section is organised as a collection of helper fuctions, each processing a part of the input, and all of which are invoked in a main-loop that identifies the boundaries between them. When the main loop returns, the call-graph is already built, and populated.

⟨ *Methods to parse text-rendering and create initial call-graph structures* 41a ⟩ ≡
⟨ *Class declaration Processor* 42 ⟩
⟨ *Method Header Processor* 43 ⟩
⟨ *Method Body Processors* 51, … ⟩
⟨ *Parameters, and return-type Encoder* 44 ⟩
⟨ *Main loop to parse text-rendering and create call-graph* 41b ⟩
◇

Fragment referenced in 34a.

The main loop reads input from the class-variable `textIn`, and identifies te points where the the class-header, method-header begin, and call their processsing methods respectively. The 'class' definition begins with the word class, and has an open brace on the same line. We use this as a signature to detect the begining of a class definition. The Method definitions have an open and close parenthesis on the same line as their descriptor. The class-constructor-common class-sstatic initializer starts with a "static " on a line by itself. These 'signatures' are recongnised and control transferred appropriately, below:

⟨ *Main loop to parse text-rendering and create call-graph* 41b ⟩ ≡

```
public static void
readInputNdCreateCallGraph (String[] args) throws java.lang.Exception {
    classListGivenUsage(args);
    runJavapInvoker();
    Thread.currentThread().sleep (2000); // milliseconds
    while (textIn != null)  {
        Thread.currentThread().yield();
        synchronized (textIn)  {
            String line;
            while ((line=textIn.readLine()) != null)  {
                if (line.indexOf('#')<0 && line.indexOf('=')<0 && line.indexOf("c
                    String cNm = readClassName (line);
                    while (! (line=textIn.readLine()).equals("{"))
                        ;
                    System.err.println (/* "Reading contents of class: " + */ cNm
                    while ((line=textIn.readLine()) != null)  {
```

```
                        line = line.trim();
                        int openP, closeP;
                        if (line.length() == 0)       continue;
                        if (line.startsWith("}"))     break; // reached end-of-cl
                        openP  = line.indexOf("(");
                        closeP = line.indexOf(");");
                        if (line.indexOf('=')<0 && line.indexOf('#')<0 &&
                            line.indexOf(':')<0 && openP>=0 && closeP>=0 && openP
                            // System.out.println (line);  // 2 be deleted
                            String mNm = readMethodHeader(cNm, line);
                            // System.out.println (cNm + ":" + mNm);  // to be de
                            if (! ((JavaBlob)exploredCode.get(mNm)).isNative  &&
                                ! ((JavaBlob)exploredCode.get(mNm)).isAbstract) {
                                ((JavaBlob)exploredCode.get(mNm)).processMethodBo
                                if (line.indexOf("   throws ") >= 0) {
                                    assert (textIn.readLine().indexOf("  Exceptio
                                    assert (textIn.readLine().indexOf("   throws
                                }
                            }
                        }  else if (line.indexOf("static ();") >= 0)
                            while ((line = textIn.readLine()) != null  &&  line.t
                                ;
                    }
                }
            }
        }
        Thread.currentThread().sleep (1000); // milliseconds
        // Thread.currentThread().yield();
    }
}
◇
```

Fragment referenced in 41a.

Above the current thread is made to `sleep` so as to afford the opportunity for the `javap`-invoker-thread to populate the shared-variable `textIn`. Similarly, later, it is made to `yield` so as to allow the other thread to re-populate it.

In the two methods defined below, the relevant details available in the class-eader, and the mothod-headers are read and recorded.

⟨ *Class declaration Processor* 42 ⟩ ≡

```
static String readClassName (String line)  {
    assert (line.indexOf("class ")>=0); // Originally  &&  line.indexOf("{")>=0);
    boolean isAbstract = line.indexOf("abstract ") >= 0;
    int     indExtends = line.indexOf(" extends ");
    String  className  = null, pName = null;
    // System.out.println (line);                  // 2 be deleted
    if (indExtends > 0)  {
```

```
            className = line.substring(line.indexOf(" class ")+7, indExtends);
            line  = line.substring (indExtends + " extends ".length());
            pName = line.indexOf(" ")>0 ? line.substring(0, line.indexOf(' ')) : line
        } else {
            line  = line.substring (line.indexOf(" class ") + " class ".length());
            className = line.indexOf(' ')>0 ? line.substring(0, line.indexOf(' ')) :
            pName= null;
        }
        className = className.replace('.', '/');
        deadlockPrediction.CodeBlob.inheritanceChain.put (className.intern(),
                                    pName == null ? null : pName.replace('.', '/')
        if (isAbstract)
            abstractClasses.add (className);
        return className;
    }
    ◇
```

Fragment referenced in 41a.

⟨ *Method Header Processor* 43 ⟩ ≡

```
    static String readMethodHeader (String cName, String line)
        throws Exception {
        int openP   = line.indexOf("("),  closeP = line.indexOf(");");
        String mNm, decoration = "", returnType = null;

        int lastBlankB4OpenP = line.substring(0, openP).lastIndexOf(" ");
        mNm = 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()
        String paramTypes = line.substring(openP+1, closeP).trim();
        int    count      = paramTypes.length() > 0  ?  1  :  0;
        for (int kk = paramTypes.length(); --kk >= 0;  )
             if (paramTypes.charAt(kk) == ',')   count++;
        String[] pTypes = new String[count];
        if (count > 0)  {
            for (int c = 0;  c < count-1;  c++)  {
                 int jj = paramTypes.indexOf(",");
                 pTypes[c]  = paramTypes.substring (0, jj);
                 paramTypes = paramTypes.substring(jj+1);
                 jj = paramTypes.indexOf(",");
            }
            pTypes[count-1] = paramTypes;
        }
        // String[] pTypes   = line.substring(openP+1, closeP).split(",");
        // System.out.println ("Params: "+line+" broken into " + pTypes.length + " co
        // for (int j = 0;  j < pTypes.length;  j++)
```

```
        //     System.out.println (pTypes[j]);

        if (cName.replace('/', '.').equals(mNm))
            mNm = "\"<init>\"";  // Encountered Constructor. ==> returnType is null.
        else  {
            // System.out.println ("cName is " + cName + "; mName is " + mNm);
            int lastBlankB4MethodName = decoration.lastIndexOf (" ");
            if (lastBlankB4MethodName < 0)  {
                returnType = decoration;
                decoration = "";
            }  else  {
                returnType = decoration.substring (lastBlankB4MethodName + 1);
                decoration = decoration.substring (0, lastBlankB4MethodName);
            }
        }
        return newJavaBlob (cName, mNm, getPRTcode(pTypes, returnType),
               decoration.indexOf("static")>=0, decoration.indexOf("synchronized")>=0
               decoration.indexOf("public")>=0, decoration.indexOf("native")>=0,
               decoration.indexOf("abstract")>=0);
    }
    ◇
```

Fragment referenced in 41a.

The parameter-list, and the return-type of a method header are processed by the utility function: `getPRTcode`, invoked above and defined below, to fetch the type-signature for a method.

⟨ *Parameters, and return-type Encoder* 44 ⟩ ≡

```
    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;
    }
```
◇

Fragment referenced in 41a.

1.8.3 Processing the method body

Recollect from the discussion in the previous section describing the cycle-detection algorithm, that there are pottentially two levels of detail as which a given method-body would require investigation: either at a coarse-level wherein only the list of methods it invokes are of interest, or at a more deteiled level, if the `monitor-enter/exit` byte-codes are invoked therein. In the interest of simplicity of implementation, and speed of execution, we proceed as below to design our implementation of this processing step:

1. Before we proceed to process the definition of the method-behaviour, its presence is confirmed by the occurrance of the **Code:** label, on a line by itself. If the '-l' option is provided when invoking 'javap', the 'LineNumberTable:' is printed before the method-behaviour is output. (Surprisingly it is output again at the end of the method-behaviour (followed by theException-table if present)). We ignore both these occurrances. The reason we supply the '-l' option while invoking 'javap', is so that we fetch the number of locals in the local-variable-array. This is printed in a line immediatetly following the "Code:" line.

2. The end of the method body is detected by the occurrance of the "Exception table:", or the "LineNumberTable:". The exception-table is optional, and it itself ends with a blank-line. The contents of the exception-table are relevant to the analysis as elaborated below. The "LineNumberTable:" is not relevant to our analysis at its current level of rigour.

3. The text-rendering of the method body is a listing of the byte-codes in the following format:

   > int-value: single-tab byte-code [arguments-followed-by-;]? [significant-comment-preseeded-by-//]?

   Every byte-code instruction is ascii-printed on a seperate line; the *int-value* above corresponds to the index into the binary byte-code array, that is immediately followed by a *cologn* and a *tab*; the text-rendering of the byte-code then appears. All of this is always present, for every byte-code. It is optionally followed by the list of arguments, which is concluded by a semi-cologn. Finally there may be a comment that is preseeded by a "//" comment-starter. We call these comments "significant" because we use their contents.

4. Depending upon whether it is sufficient to examine the method-body at a coarse-level, or that a more detailed analysis is required, our implementation does a single pass, or a two-pass analysis of the method-body, respectively. In the first pass, we read the text-rendering off the input-stream, and create a couple of Arrays (called, `offset`, and `code`) to record its contents: for every line of the method-body, its offset, and the rest of the code on it is stored in $<$ `offset`, `code` $>$. While doing so if it is noticed that the input method-body doesnot use the `monitor-enter/exit` byte-code, then at the end of the method-body, the single pass that just concluded is sufficient, else the second pass kicks-in which starts its analysis from the data stored in the arrays.

5. During the first pass, we need to detect, and process the ocurrance of three tokens belonging to the *invoke* family of op-codes, that correspond to an invocation of another method: *invokevirtual*, *invokespecial*, and *invokestatic*. Yet another member of this family, namely the *invokeinterface* byte-code, is ignored since we are (currently) discounting the polymorphism feature of Java. Notice that wherever any of these byte-codes is invoked, there appears a 'significant-comment' as the last thing on that line. This significant comment contains the name of the method to be invoked. At such points the `noteCallTo` method of the parent `CodeBlob` class is invoked to create edges of the call-graph.

6. If during the first pass any monitor instructions are not encountered, then $<$ `offset`, `code` $>$ is cleared, and the method returns having completed its analysis, and created all the edges of the call-graph that it had to. If, however that is not the case, then the execution of the method continues into the second-pass as below.

7. During the second pass,the two things that need to be found and recorded is the type of the object used as a monitor, and the last *matching* `monitorexit` for a given `monitorenter` byte-code. The code enclosed between every such pair of a `monitorenter` and its last matching `monitorexit` is represented by a seperate `CodeBlob` node in our graph. These additional nodes are created during the second-pass, and so also the caller/called edges of the call-graph corresponding to method-invocations enclosed in such code are created in the second-pass. Invocations of `noteCallTo` during the first pass are therefore guarded, additionally, by the value of the boolean-flag that records te occurrance of the `monitor`-pair of instructions.

8. As a part of the second-pass, we instantiate, and compute a `typeAtTOSlist` that records the type-of-the-object at the top-of-the-operand-stack just prior to the execution of every byte-code of the method-body. Like there is the (single copy of) operand-value-stack that is used in the execution of the byte-code, wherein the stack-depth and the values therein keep getting updated as execution proceeds, we maintain an operand-*type*-stack to populate the `typeAtTOSlist` wherein the type of the object on top of the operand-stack is recorded for every byte-code in the method-body. (just prior-to-execution of the byte-code). Said differently, we *interpret* (at the level of types) the execution of the byte-code to

discover the type of the object that will be at the top-of-operand-stack just prior to the execution of every byte-code. In order to run this simulation we need to maintain the state of the `typeStack`, and the `typeArray` that, respectively, record the *types* of the objects on the operand-stack, and the local-variable-array, as the simulation of the byte-code unfolds. At the begining of the method-body-simulation, the `typeStack` is initialized to be empty, and the argument-list of the method is used to initialize the first few slots of the `typeArray`.

9. Among the two peices of information we require: the type recorded in the `typeAtTOSlist` associated with the `monitorenter` byte-code tells us the type of the monitor it attempts to lock. The other peice of information, namely the *matching* `monitorexit` instruction, is encoded into a corresponding pair of entries in the "Exception table:". For instance, the code pattern: "`synchronized (` *expression* `) this.doThat();`" is translated by `javac` into byte-code which when rendered into text by `javap` looks like follows:

```
 ...:  ...
  13:  aload_3
  14:  monitorenter
 ...:  ...; // code corresponding to invocation of this.doThat();
  23:  aload_3
  24:  monitorexit
  25:  astore_2
  26:  aload_3
  27:  monitorexit
  28:  aload_2
  29:  athrow
 ...: ...
Exception table:
    From To Target Type
      15 25 25 any
      25 28 25 any
```

Notice in the above byte-code-sequence, there is a single `monitorenter` but two `monitorexit`s. The first one corresponds to the release of the monitor after normal execution of `this.doThat()`, and the second one corresponds to the release of the monitor in case an exception occurrs during the attempt to compute `this.doThat()`. So also, notice that the "Exception table:" has two entries that have '25' as the target, and 'any' as their 'Type'. The first of them encapsulates the entire body of the `synchrinized` statement between its 'from' and 'to' byte-code offsets. This is the entry that we can use to identify the *matching* `monitorexit` corresponding to every `monitorenter` instruction. More specifically: the 'from-value' corresponding to this entry is always one greater than the byte-code-offset of the monitor-enter that marks the begining of the encapsulated-code, and the 'to-value' is one greater than the byte-code-offset corresponding to the encapsulated-code-end. Occassionnally, when there

is an 'if-then-else' statement as the encapsulated-code, wherein on one of the branches (either the then-branch or the else-branch) the processing terminates with a return of control to the calling method, then there can be more then two `monitorexit` instructions corresponding to the `monitorenter`. All such entries in the "Exception table:" (except the last one of them) would correspond to the 'normal-preocessing' od the'encapsulated-code'.

The code that loops over the method-body maintains a (pair of) stack(s) that is updated whenever monitor instructions are encountered. Stack being a last-in-first-out data-structure, is appropriate since that is the only way monitors can be used correctly in Java/C#.

10. To further understand how the *matching* `monitorexit` is identified using the entries in the "Exception table:", let us look at an interesting specimen of the said table.

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

 155   162   165   Class java/lang/InterruptedException

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

From the above table, we firstly discount all entries that have the same value for the 'From' and the 'To' column, while having 'any' in the 'Type' column. We do this since the byte-codes encapsulated by the do-dropped entries are the ones introduced by the compiler, and do-not correspond to any source, per-se. Moreover, the dropped code typically deosnot 'invoke' any other-functionality, so nothing lost, really. That fetches us the following table:

```
Exception table:
 from   to  target type
   6    54    65   any
```

```
 55    62    65   any
 88    99   108   any
102   105   108   any
116   123   126   Class java/lang/InterruptedException

155   162   165   Class java/lang/InterruptedException

148   173   176   any
192   202   205   any
 79   227   230   any
LineNumberTable:
 ...
```

The next step of processing merely registers (into `executionStartOffsets`) the fact that 126, and 165 are valid control-transfer-targets, therafter also discarding the corresponding entries from the above table to fetch the one below:

```
Exception table:
 from   to  target type
   6    54    65   any
  55    62    65   any
  88    99   108   any
 102   105   108   any
 148   173   176   any
 192   202   205   any
  79   227   230   any
LineNumberTable:
 ...
```

The next step of processing then verifies that all entries corresponding to the same value of 'target' do indeed refer to 'contiguous-blocks-of-byte-code' using the information in the `byteOffset` List. Having ascertained this it then reduces the table above to the one below that it then retains in the form of a Map called `matchingExits`.

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

11. In creating the `typeAtTOSlist`, an additional level of abstraction is employed to further simplify the implementation: since we are interested in the types of the reference objects that are used by the *monitor-enter/exit* byte-codes, we can club all other basic types (namely, `int`, `long`, `short`, `byte`, `char`, `float`, and `double`) in the byte-code notation into one, called the non-reference-type.

However, when the operand is indeed a reference, we store the actual java-type of the reference. Conducting the analysis at this level of abstraction suffices our purpose.

12. It is important to notice that the VM-Spec. discusses the layout of the operand-stack and its use in the execution of the byte-code in terms of "words", while also maintaining that illegal attempts to look at two contiguous words (on the stack) as a single long/double (by a byte-code instruction) are detected by the byte-code-verification step and prohibited from execution. Moreover, it distinguishes the basic arithmatic-types, into *category 1 & 2*, based upon whether they are single-word, or double-word, respectively. This categorisation is later refered in the implementation of stack-manipulation instructions like dup2_w, for instance, that have a context-sensitive effect on the contents of the stack. We shall therefore interpret the evolution of the operand-stack at the (abstraction) level of *words*.

13. The only two multi-line byte-codes are *tableswitch* & *lookupswitch*, because their jump-targets table is printed with one entry-per line. Typically these instructions end with the '*default:*' token, and its jump-target on the last line.

14. Correction for calls to the `clone` functionality: After executing the program under development, it was observed that 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:()Ljava/lang/Object; This is handled by replacing such nodes by references to the already created node corresponding to the `java.lang.Object.clone`.

15. Given that we are *simulating* the execution of the byte-code at the abstraction-level of *types-of* operands & variables, and moreover given our assumption that the byte-code is well-formed, we need conduct the simulation only at a level of detail that is sufficient for us to discover the information we need, modelling as little of the detail as is relevant, and abstracting all else. The following way to maintain the values of `typeStack`, and `typeArray` seems sufficient: if the operand/object is of one of the single-word basic types (char, byte, short, int, or float), we represent it with a *"c1"*; if it is one of the double-word basic types (long or double), we represent it with a *"c2"*; if the operand/object is a reference, then the *type-name* of the reference is recorded; if the operand/object is an array of objects its representation-string starts with as-many of '[' symbols as is the dimention of the array, and followed by: *"c1"* to represent any of the basic types, or a *"c2"*, like above, whereas if it is an array of objects of type reference, then the *type-name* of the elements of the array is recorded. Finally, the `typeAtTOSlist` entry corresponding to every instruction is populated by merely reading off the top-of-the-`typeStack`.

16. While simulating the execution of the byte-code as described above, special care is required when handling control-transfer instructions: the states of the two variables `typeStack`, and `typeArray` are to be the same for all pottential control-transfer targets of a control-transfer byte-code. In order to program for

this requirement our implementation maintains a three-tupple: <`executionStartOffset`, `stackAtExecutionStart`, and `arrayAtExecutionStart`>. On encountering control-transfer instructions the lists in the three-tupple are appended as a part of simulating their execution, with the control-transfer targets, and the states of the operand-stack, and the local-variables-array at that point.

17. In the second-pass over the method-body, which is conducted using <`offset`, `code`>, the actual effects on the `typeStack`, and `typeArray`, of executing an instruction is affected by the `interpret` method. Clearly this method is invoked in a loop, every-time handing it the next instruction to interpret. However this loop-execution continues only so far as the `interpret` method continues to return `true`. For instructions like *[adfil]return* it returns `false`, to indicate that the execution trail terminates here, and the next instruction in the sequence can-not be analysed assuming the operand-stack-state as the current instruction would leave it. Similarly `interpret` returns `false` for *goto(_w), jsr(_w), ret, lookupswitch, tableswitch & return*. For all other control-transfer instructions, the `interpret` does return `true`.

18. The analysis of the byte-code corresponding to the method-body therefore happens in parts, starting from the begining or from the target of some control-transfer instruction, and going-on until the `interpret` method returns `false`. Every-time this inner loop terminates, the next target of some control-transfer instruction is read from the `executionStartOffset`, and the `typeStack` is accordingly initialised. Similarly for the `typeArray`. And the loop so initialised continues the analysis of the corresponding portion of the method-body.

19. It is appropriate to mention here that *return-address* is a type. This type is not exported to the java-programmer, but is available at the byte-code level. For instance the *jsr / jsr_w* byte-codes push the offset of the following instruction on the operand-stack before transfering control to the address available as its operand. The *ret* byte-code transfers control *back* to the return-address stored in the local-variables-array at the index identified by its operand. In our modelling of the `typeStack`, and the `typeArray`, we model *return-address* by recording and reproducing its actual value.

⟨ *Method Body Processors* 51 ⟩ ≡

```
public void processMethodBody(String cbn) throws java.lang.Exception {
    List offset = new ArrayList(),
         code   = new ArrayList();
    boolean monitorUsed = false;
    String line = textIn.readLine(), tl = null;         // === ditto ===
    // System.out.println (line);                          // to be deleted
    if (line.startsWith("  Synthetic:"))
        line = textIn.readLine();
    assert (line.startsWith("  Code:"));                 // === ditto ===
    assert ((line = textIn.readLine()) != null  &&  line.trim().length() > 0);
    // System.out.println (line);                          // to be deleted
```

```
int locals = Integer.parseInt (line.substring (line.indexOf("Locals=")+"Local
                                               line.indexOf(", Args_")));
List executionStartOffset  = new ArrayList(); // Program-Counters where execu
List stackAtExecutionStart = new ArrayList(); // Stack-state at PC where exec
List arrayAtExecutionStart = new ArrayList(); // Local-state at PC where exec
Integer m1 = new Integer (-1);
int bcOffset, colon = -1;
List matchingMonitorExits = new ArrayList();

while ((line=textIn.readLine()) != null  &&  line.trim().length() > 0)  {// U
    if (line.indexOf("Exception table:")>=0)  {  // point 1 above.
        if (! monitorUsed)  {
            while ((line = textIn.readLine()) != null  &&  line.indexOf("Line
                ;
            while ((line = textIn.readLine()) != null  &&  line.trim().length
                ;
            break;
        }
        int sz = readExceptionNdLineNoTable (executionStartOffset, stackAtExe
                                             offset, code, matchingMonitorExi
        for (  ;  --sz >= 0;  )
            arrayAtExecutionStart.add (getTypeArrayFromArguments (cbn, locals
        break;
    } else if (line.indexOf("LineNumberTable:") >= 0) {
        while ((line = textIn.readLine()) != null  &&  line.trim().length() >
            ;
        break;
    }
    // System.out.println (line);                       // to be deleted
    assert ((colon = line.indexOf(':')) > 0);
    bcOffset = Integer.parseInt (line.substring(0, colon).trim());
    if (line.indexOf("tableswitch{")>=0 || line.indexOf("lookupswitch{")>=0)
        while((tl = textIn.readLine()).indexOf("default:")<0)
            line = line + tl;                        // Multi-line instruction
    line = line.substring(colon+1).trim();           // instruction, operands,
    offset.add (bcOffset);                           // points 3,
    code.add   (line);                               //   and 4 above.
    if (line.startsWith("monitore"))
        monitorUsed = true;                          // point 6 above
    if (! monitorUsed &&                             // point 7 above
        (line.startsWith("invokevirtual\t") ||       // point 5 above
         line.startsWith("invokespecial\t") ||       // -- " --
         line.startsWith("invokestatic\t"))) {       // -- " --
        String ms = line.substring (line.indexOf("; //Method ") + "; //Method
        String mn = ms.substring (0, ms.indexOf(':'));
        if (mn.charAt(0)=='\"'  &&  mn.endsWith("\".clone"))             //
            noteCallTo (cbn, "java/lang/Object.clone:()Ljava/lang/Object;");
        else {
            String cn = cbn.substring (0, cbn.lastIndexOf("."));
            noteCallTo (cbn, (mn.indexOf('.')<0 ? (cn.replace('.', '/') + "."
```

```
            }
        }
    }
    // System.out.println ("Completed reading method: " + cbn);
    if (! monitorUsed)  {                                // Second pass is unneces
        offset.clear();  code.clear();    executionStartOffset.clear();
        stackAtExecutionStart.clear();    arrayAtExecutionStart.clear();
        return;                                         // per point 6, above. Cle
    }
    // System.out.println ("Detected usage of monitor:" + cbn);  // To be deleted
    java.lang.Object[] typeAtTOSlist = new java.lang.Object[offset.size()];

    // PC-Zero is where Execution begins, first with an empty stack, per point 8,
    updateBBlist (0, new Stack(),getTypeArrayFromArguments (cbn, locals), offset,
                  executionStartOffset, stackAtExecutionStart, arrayAtExecutionSt

    Stack objStack = new Stack(), nameStack = new Stack();
          objStack.push (this);   nameStack.push (cbn);
    StringBuffer enterCounter = new StringBuffer("0");
    for (int i = 0;  i < executionStartOffset.size();  i++)  {
        bcOffset           = ((Integer)executionStartOffset.get(i)).intValue();
        Object[] typeArray =  (Object[]) arrayAtExecutionStart.get(i);
        Stack typeStack =  (Stack) stackAtExecutionStart.get(i);
        if (offset.contains(bcOffset)) {
            // System.out.println ("Execution-start-offsets: " + executionStartOf
            // System.out.println ("Starting interpretation from offset: " + bcOf
            for (int j = offset.indexOf(bcOffset);  j<offset.size();  j++)  {
                typeAtTOSlist[j] = typeStack.empty() ? null : typeStack.peek();
                // System.out.println ("Before: "+offset.get(j)+ ' ' +typeStack +
                offset.set (j, m1);
                if (! interpret(objStack, nameStack, (String) code.get(j), typeSt
                            executionStartOffset, stackAtExecutionStart, arrayAtE
                            j == offset.size()-1 ? -1 : ((Integer)offset.get(j+1)
                            enterCounter, offset, matchingMonitorExits))
                    break;
            }
        }
    }
}
public int readExceptionNdLineNoTable (List execStart, List stacks, List offset,
                                       List matchingExits, String cbn) throws jav
    assert (textIn.readLine().trim().startsWith("from"));
    int    i[] = new int[15], j[] = new int[15], k[] = new int[15], t=-1, n=0, ni
    String l = textIn.readLine();
    for (;  l != null  &&  l.trim().indexOf("LineNumberTable:") < 0;  l = textIn.
        if (l != null  &&  l.trim().length() > 0)  {
            ni = Integer.parseInt ((l = l.trim()).substring (0, t = l.indexOf(' '
            l  = l.substring(t).trim(); // read and dropped the 'from' integer va
            nj = Integer.parseInt (l.substring (0, t = l.indexOf(' ')));
            l  = l.substring(t).trim(); // read and dropped the 'to' integer valu
```

```
                nk = Integer.parseInt (l.substring (0, t = l.indexOf(' ')));
                l  = l.substring(t).trim(); // read and dropped the 'target' integer
                if (l.startsWith("any"))  {
                    if (ni != nk  &&  ((String)code.get(offset.indexOf(nj)-1)).starts
                       i[n] = ni;  j[n] = nj;  k[n] = nk;  n++;
                       // System.out.println ("Processed: " + ni +','+ nj +','+ nk +'
                    }
                } else {
                    // execStart.add (nk);
                    // Stack s = new Stack();
                    // s.push ("L" + l.substring(l.indexOf(' ')+1) + ";");
                    // stacks.add (s);
                }
            }
        }
        for (int m = 0;  m < n;  m++)
            if (m != (n-1)  &&  k[m] == k[m+1]) {
                if ((offset.indexOf(new Integer(j[m]))+1) != offset.indexOf(new Integ
                    System.out.println ("Proceeding with Inaccurate analysis of " + c
                i[m+1] = i[m];
            } else if (i[m]!=0 && ((String)code.get(offset.indexOf(i[m])-1)).startsWi
                matchingExits.add (j[m]);
        // System.out.println ("Matching Exits at n(" + n + "): " + matchingExits);
        while ((l=textIn.readLine()) != null  &&  l.trim().length() > 0)
            ;
        return execStart.size();
    }
    ◇
```

Fragment defined by 51, 55, 61.
Fragment referenced in 41a.

Since the `interpret` method is a static-method, the `this` pointer in its body cannot be used to identify the node on which it operates. The first two arguments passed to its invocation are therefore objects of type `Stack`. The first is an object-stack that contains `CodeBlob` objects, and the second is a stack of names corresponding to the objects on the object-stack. This method creates a new `CodeBlob` object when encountering a `monitorenter` instruction, and pushes it on-top-of the object-stack. its corresponding name is pushed on-top-of the nameStack. On encountering the last matching `monitorexit` instruction, it pops the corresponding object off the top-of-objectStack, and similarly its name off the nameStack. The `interpret` method *interprets* one instruction per invocation, that is passed as its third argument. Recall that the significance of its (boolean) return value is discussed previously. It updates its fourth and fifth arguments which represent the operand-type-stack, and the local-variables-array, respectively, to record the effect of having *abstract-interpretted* the input instruction. If the input instruction is a control-transfer instruction, the next three invocation-arguments are updated, if necessary. They are appended with the control-transfer target of the instruction iff they do-not already contain that target-address, and

the corresponding stack-state & local-variables-array-state in which execution starts there, respectively. The next two arguments are integers: one that gives the offset of the next instruction in the sequence, and another that counts-up every-time the bytecode is an occurrance of `monitorenter`. The last argument is a `stack`. It is used to record the *farthest jump target* that is encountered within a `synchronized` block of statements. This stack has always the same number of entries as are in the objStack or the nameStack.

1. The implementation below uses the `java.util.regex` API from the J2SE library. Various objects of type `Pattern` are created to match (sub)sets of instructions in the *input*, and based upon that, accordigly the states of the `typeStack` and the `typeArray` are updated to reflect its *abstract-interpretation*.

2. The *invoke* family of instructions are to be *interpreted* by consuming the appropriate number of *operands* from the operand-type-stack, given the signature of the method invoked, and also with due consideration to whether it is an instance method or class-metod. Further, the return from such an invocation is interpreted by pushing onto the operand-type-stack its return-type. A helper method, `interpretInvocation` is used to affect this. The `interpret` method invokes `noteCallTo` to update the caller/called edges of the call-graph. Notice that the `processMethodBody` method discontinues invocations of `noteCallTo` method upon encountering a `monitorenter` instruction. For methods that contain `synchronized` block of code, the task of populating the corresponding `CodeBlob` objects' `calledCode`/`callingCode` attributes is carried out by the `interpretInvocation` method. Similarly, also notice that on encountering the `monitorenter` instruction `interpret` creates a new node of the call-graph, per points 5, 6, 7 above.

⟨ *Method Body Processors* 55 ⟩ ≡

```
// import java.util.regex;
static Pattern u_arith = Pattern.compile ("[ilfd](?:neg)");           // unary ne
static Pattern dtc     = Pattern.compile ("[dfil]2[bcsdfil]");        // Data Typ
static Pattern wideinc = Pattern.compile ("(?:wide)|(?:iinc)");       // wide / i
static Pattern loads   = Pattern.compile ("[adfil]load(?:_[0123])?"); // Loads
static Pattern stores  = Pattern.compile ("[adfil]store(?:_[0123])?");// Stores
static Pattern consts  = Pattern.compile ("[dfil]const_m?[012345]");  // Push Con
static Pattern push    = Pattern.compile ("[bs]ipush");               // byte/sho
static Pattern b_arith = Pattern.compile ("[ilfd]((?:add)|(?:sub)|(?:mul)|(?:div)
static Pattern bitwise = Pattern.compile ("[il]((?:x?or)|(?:and))");  // bitwise:
static Pattern shifts  = Pattern.compile ("[il]u?sh[lr]");            // Shift ri
static Pattern lfdCmp  = Pattern.compile ("[lfd]cmp[lg]?");           // double-w
static Pattern cmpares = Pattern.compile ("if(?:_[ai]cmp)?((?:eq)|(?:g[et])|(?:l[
static Pattern nullCmp = Pattern.compile ("if(?:non)?null");          // Null com
static Pattern returns = Pattern.compile ("[adfil]?ret(?:urn)?");     // return f
static Pattern aloads  = Pattern.compile ("[abcdsilf]aload");         // Array Lo
static Pattern astores = Pattern.compile ("[abcdsilf]astore");        // Array St
static Pattern gotoJsr = Pattern.compile ("(?:goto)|((?:jsr)(?:_w)?)"); // goto/j
static Pattern mTarget = Pattern.compile ("((lookup)|(table))(switch)"); // multi
```

```
static Pattern ldConst = Pattern.compile ("ldc2?(?:_w)?");           // Load con
static Pattern stackOp = Pattern.compile ("(?:dup2?(?:_x[12])?)|(?:pop2?)|(?:swap
static Pattern invokes = Pattern.compile ("invoke((special)|(static)|(virtual)|(i
static Pattern fieldOp = Pattern.compile ("((?:get)|(?:put))((?:field)|(?:static)
static Pattern castOp  = Pattern.compile ("(?:checkcast)|(?:instanceof)");
static Pattern newOp   = Pattern.compile ("(?:multi)?a?new(?:array)?");  // vario
static Pattern lockOp  = Pattern.compile ("monitor((?:enter)|(?:exit))");  // Mon
static Pattern restOp  = Pattern.compile ("a((?:throw)|(?:rraylength)|(?:const_nu
static Pattern c1Match = Pattern.compile ("[BCFISZ]");               // c1 match
static Pattern c2Match = Pattern.compile ("[DJ]");                   // c2 match
static String  c1      = "c1".intern(),
               c2      = "c2".intern();

static boolean interpret(Stack oStack, Stack nStack, String bc, Stack tS, Object[
    List execStartOffset, List stackAtExecStart, List arrayAtExecStart,
    int nextOffset, StringBuffer eCounter, List bcOffsets, List monitorExits) thr
    Matcher m; // System.out.println (bc);
    if (bc.startsWith("nop") || u_arith.matcher(bc).lookingAt())
        ;
    else if ((m = dtc.matcher(bc)).lookingAt())  {
        char from = bc.charAt(m.start()), to = bc.charAt(m.end()-1);
        if (from == 'd'  ||  from == 'l') { assert (tS.pop() == c2);  assert (tS.
        else                              assert (tS.pop() == c1);
        if (to == 'd'  ||  to == 'l')     { tS.push(c2);  tS.push(c2); }
        else                              tS.push(c1);
    } else if ((m = loads.matcher(bc)).lookingAt()  ||  (m = stores.matcher(bc)).
               (m = wideinc.matcher(bc)).lookingAt())  {
        if (bc.charAt(m.start()) == 'w')    bc = bc.substring(m.end()).trim();
        if (bc.startsWith ("iinc"))         return true;
        if ((m = loads.matcher(bc)).lookingAt()  ||  (m = stores.matcher(bc)).loo
            char tb = bc.charAt(m.start()), tc = bc.charAt(m.end()-1);
            int i = (tc!='d' && tc!='e') ? (tc-'0') : Integer.parseInt(bc.substri
            if (bc.charAt(m.start()+1) == 'l')
                if (tb == 'd'  ||  tb == 'l') {
                    assert (tA[i] == c2);      assert (tA[i+1] == c2);
                    tS.push(c2);               tS.push(c2);
                } else
                    tS.push (tA[i]);
            else
                if (tb == 'd'  ||  tb == 'l') {
                    assert(tS.pop() == c2);   assert(tS.pop() == c2);
                    tA[i] = c2;               tA[i+1] =  c2;
                } else // if (tA[i]==null || !((String)tS.peek()).equals("Ljava/l
                    tA[i] = tS.pop();
                // else
                    // tS.pop();
        }
    } else if ((m = consts.matcher(bc)).lookingAt()  ||  (m = push.matcher(bc)).l
        char tb = bc.charAt(m.start());
        if (tb == 'd'  ||  tb == 'l') { tS.push(c2);  tS.push(c2); }
```

```
    else                            tS.push(c1);
} else if ((m = b_arith.matcher(bc)).lookingAt() || (m = bitwise.matcher(bc))
    char tb = bc.charAt(m.start());
    if (tb == 'd'  ||  tb == 'l') {
        assert(tS.pop() == c2);   assert(tS.pop() == c2);
        assert(tS.pop() == c2);   assert(tS.pop() == c2);
        tS.push(c2);              tS.push(c2);
    } else {
        assert(tS.pop() == c1);   assert(tS.pop() == c1);    tS.push(c1);
    }
} else if ((m = shifts.matcher(bc)).lookingAt()) {
    assert (tS.pop() == c1);
    if (bc.charAt(m.start()) == 'l') {
        assert(tS.pop() == c2);   assert(tS.peek() == c2);   tS.push(c2);
    } else
        assert(tS.peek() == c1);
} else if ((m = lfdCmp.matcher(bc)).lookingAt()) {
    char tb = bc.charAt(m.start());
    if (tb == 'd'  ||  tb == 'l') {
        assert(tS.pop() == c2);   assert(tS.pop() == c2);
        assert(tS.pop() == c2);   assert(tS.pop() == c2);
    } else {
        assert(tS.pop() == c1);   assert(tS.pop() == c1);
    }
    tS.push(c1);
} else if ((m = cmpares.matcher(bc)).lookingAt() || (m = nullCmp.matcher(bc))
    tS.pop();
    if (bc.charAt(m.start()+2) == '_')
        tS.pop();
    int ctt = Integer.parseInt (bc.substring (m.end()).trim()); //control tra
    updateBBlist (ctt, tS, tA, bcOffsets, execStartOffset, stackAtExecStart,
} else if ((m = returns.matcher(bc)).lookingAt())  {
    char tb = bc.charAt(m.start());
    if (tb=='d' || tb=='l')           {  assert(tS.pop() == c2);   assert(tS.
    else if (tb=='a')                    tS.pop();
    else if (m.group().length() == 7)    assert(tS.pop() == c1);
    else if (m.group().length() == 3) {
        int ctt = ((Integer) tA[Integer.parseInt(bc.substring(m.end()).trim()
        updateBBlist (ctt, tS, tA, bcOffsets, execStartOffset, stackAtExecSta
    }
    return false;
} else if ((m = aloads.matcher(bc)).lookingAt()) {
    char tb = bc.charAt(m.start());
    assert (tS.pop() == c1);                                              // c
    if (tb != 'a') {       assert (((String)tS.pop()).startsWith("["));  // c
        if (tb=='d' || tb=='l') {  tS.push(c2);         tS.push(c2);  }
        else                       tS.push(c1);
    } else {
        String tmp = (String) tS.pop();
        assert (tmp.startsWith("[") || tmp.equals("Ljava/lang/Object;")); //
```

```
        tS.push(tmp.startsWith("[") ? tmp.substring(1) : tmp); // consume the
    }
} else if ((m = astores.matcher(bc)).lookingAt())  {
    char tb = bc.charAt(m.start());
    if (tb == 'a')                  tS.pop();
    else if (tb=='d' || tb=='l') {  assert(tS.pop() == c2);   assert(tS.pop()
    else                            assert(tS.pop() == c1);
    assert(tS.pop() == c1);         assert(((String)tS.pop()).startsWith("[")
} else if ((m = gotoJsr.matcher(bc)).lookingAt())  {
    if (bc.charAt(m.start()) == 'j')
        tS.push(nextOffset);
    int ctt = Integer.parseInt (bc.substring (m.end()).trim());
    updateBBlist (ctt, tS, tA, bcOffsets, execStartOffset, stackAtExecStart,
    return false;
} else if ((m = mTarget.matcher(bc)).lookingAt()) {
    // System.out.println ("entered here !");
    for (int cln = bc.indexOf(':'), sCln = bc.indexOf(';');  cln<0 && sCln<0;
        int ctt = Integer.parseInt (bc.substring (cln+1, sCln).trim());
        updateBBlist (ctt, tS, tA, bcOffsets, execStartOffset, stackAtExecSta
        bc = bc.substring(sCln+1);  cln  = bc.indexOf(':');  sCln = bc.indexO
    }
}  else if ((m = ldConst.matcher(bc)).lookingAt())
    if (m.group().length() > 3  &&  bc.charAt(3) == '2') {
        tS.push (c2);  tS.push (c2);
    } else {
        String tmp = bc.substring(m.end());
        tS.push(tmp.indexOf("//String")>=0 || tmp.indexOf("//class")>=0  ?  "
    }
else if ((m = stackOp.matcher(bc)).lookingAt())  {
    char tb = bc.charAt(m.start());
    Object ts = null, tsn = null, tsnn = null, tsnnn = null;
    if (tb == 'd') {
        if (m.group().length() == 3) {  assert ((ts = tS.peek()) != c2);  tS.
        else if (m.group().length() == 4) {                              //dup
            if ((ts=tS.pop()) == c2) {  assert (tS.peek() == c2);  tS.push(c2
            else {  assert((tsn=tS.peek()) !=  c2);  tS.push(ts);  tS.push(ts
        } else if (m.group().length()==6 && bc.charAt(m.end()-1)=='1') {  //d
            assert ((ts=tS.pop()) != c2);        assert ((tsn=tS.pop()) != c2
            tS.push(ts);         tS.push(tsn);   tS.push(ts);
        } else if (m.group().length()==6 && bc.charAt(m.end()-1)=='2') {  //d
            assert ((ts=tS.pop()) != c2);
            if ((tsn=tS.pop()) == c2) {
                assert (tS.pop() == c2);         tS.push(ts);
                tS.push(c2);     tS.push(c2);    tS.push(ts);
            } else {
                assert ((tsnn=tS.pop()) != c2);  tS.push(ts);
                tS.push(tsnn);   tS.push(tsn);   tS.push(ts);
            }
        } else if (m.group().length()==7 && bc.charAt(m.end()-1)=='1') {  //d
            ts = tS.pop();       tsn = tS.pop(); assert((tsnn=tS.pop()) != c2
```

```
            tS.push(tsn);        tS.push(ts);    tS.push(tsnn);   tS.push(tsn
        } else if (m.group().length()==7 && bc.charAt(m.end()-1)=='2') {  //d
            ts = tS.pop();       tsn = tS.pop(); tsnn = tS.pop(); tsnnn = tS.
            assert ((ts!=c2 && tsn!=c2)  ||  (ts==c2 && tsn==c2));
            assert ((tsnn!=c2 && tsnnn!=c2)  ||  (tsnn==c2 && tsnnn==c2));
            tS.push(tsn);        tS.push(ts);
            tS.push(tsnnn);      tS.push(tsnn);  tS.push(tsn);    tS.push(ts)
        }
    } else if (tb == 'p') {                      // pop or pop2
        if (m.group().length() == 3)             assert(tS.pop() != c2);
        else if ((ts=tS.pop()) != c2)            assert(tS.pop() != c2);
        else                                     assert(tS.pop() == c2);
    } else {                                     // The instruction would be
        assert ((ts = tS.pop()) != c2);          assert ((tsn = tS.pop()) !=
        tS.push(tsn);                            tS.push(ts);
    }
} else if ((m = invokes.matcher(bc)).lookingAt())
    interpretInvocation (m.group().endsWith ("static"),
        bc.substring(bc.indexOf("//Method ")+"//Method ".length()), tS,
        (String)nStack.peek(), (JavaBlob)oStack.peek());
else if ((m = fieldOp.matcher(bc)).lookingAt())  {
    Object ts = null;
    if (bc.charAt(m.start()) == 'g') {
        if (bc.charAt(m.start()+3) == 'f')
            assert ((ts = tS.pop()) != c1  &&  ts != c2);
        tS.push (abstractTenc (bc.substring(bc.indexOf(':')+1).trim()));
        if (((String)tS.peek()).equals(c2))    tS.push(c2);
    } else {
        if ((ts=tS.pop()) == c1)
            assert (c1Match.matcher(bc.substring(bc.indexOf(':')+1)).lookingA
        if (ts==c2) {            assert (tS.pop()==c2);
            assert (c2Match.matcher(bc.substring(bc.indexOf(':')+1)).lookingA
        }
        if (bc.charAt(m.start()+3) == 'f')
            assert ((ts = tS.pop())!=c1   &&   ts!=c2);
    }
} else if ((m = castOp.matcher(bc)).lookingAt())  {
    Object ts = null;
    assert ((ts=tS.pop()) != c1  &&  ts != c2);
    if (bc.charAt(m.start()) == 'i')
        tS.push (c1);
    else {
        String t = bc.substring(bc.indexOf("//class ") + "//class ".length())
        tS.push (t.charAt(0) == '"'  ?  t.substring(1, t.length()-1)  :  ("L"
    }
} else if ((m = newOp.matcher(bc)).lookingAt())  {
    String typePreFix = "";
    if (!m.group().equals("new")  &&  !m.group().startsWith("multi"))  {
        assert (tS.pop() == c1);
        typePreFix = "[";
```

```
    } else if (m.group().equals("multianewarray")) {
        int i = bc.indexOf('#');
        while (bc.charAt(i++) != ' ');
        for (i = Integer.parseInt (bc.substring(i, bc.indexOf(';')).trim());
             --i >= 0  &&  tS.peek()==c1;  tS.pop())
            typePreFix += "[";
    }
    tS.push (typePreFix + (m.group().equals("newarray")
                           ?  abstractTenc (encodeType(bc.substring(m.end()).
                           :  ("L"+bc.substring(bc.indexOf("//class ") + "//c
} else if ((m = lockOp.matcher(bc)).lookingAt())  {
    deadlockPrediction.CodeBlob po = (JavaBlob) oStack.peek();
    if (m.group().endsWith("enter")) {
        String pnm = (String) nStack.peek();
        String lock= (String)tS.pop();
        if (lock.startsWith("L"))
            lock   = lock.substring (1, lock.length()-1);// Get rid of the le
        String key = (pnm +'#'+ lock +'#'+ eCounter.charAt(0)).intern();
        oStack.push (((JavaBlob)po).newSJavaBlob(pnm, key, lock));
        nStack.push (key);
        eCounter.setCharAt(0, (char)(eCounter.charAt(0) + 1));
    } else {
        // System.out.println ("Compare: " + (String)tS.peek() + ":" + po.get
        // System.out.println ("monitorExits: " + monitorExits + "; nextOffse
        if (monitorExits.contains(nextOffset)) {
            String lock = (String) tS.peek();
            if (lock.startsWith("L"))
                assert (lock.equals ("L"+po.getLockType()+";"));
            else
                assert (lock.equals (po.getLockType()));
            oStack.pop();  nStack.pop();
        }
        tS.pop();
    }
} else if ((m = restOp.matcher(bc)).lookingAt()) {
    if (! m.group().equals("aconst_null")) {
        // System.out.println ("and not entered here !");
        Object ts = null;
        assert ((ts=tS.pop()) != c1  &&  ts != c2);
        if (m.group().equals("arraylength"))
            tS.push (c1);
        else  {
            for (int i = tS.size();  --i >=0 ;  tS.pop());
            tS.push (ts);
            return false;
        }
    } else
        tS.push ("Ljava/lang/Object;");
} else
    System.out.println ("Unprocessed instruction!!: " + bc);
```

```
        if (eCounter.charAt(0) != '0')
            ; // System.out.println ("Key: " + (String)nStack.peek()+ ";\nLock: " +
              //                   ((JavaBlob)oStack.peek()).getLockType());
        return true;
    }
    ◇
```

Fragment defined by 51, 55, 61.
Fragment referenced in 41a.

⟨ *Method Body Processors* 61 ⟩ ≡

```
    public Object[] getTypeArrayFromArguments (String cbn, int locals)  {
        // System.out.println ("Local-count Received here: " + locals+ "isStatic: " +
        java.lang.Object[] retval  = new java.lang.Object[locals];
        String pList = cbn.substring (cbn.indexOf(":")+2, cbn.indexOf(")"));
        String prefix = "";
        if (! isStatic)
            retval[0] = "L" + cbn.substring(0, cbn.indexOf('.')) + ";";
        for (int j = isStatic ? 0 : 1;  pList.length() > 0;  pList = pList.substring(
            if (pList.charAt(0) == '[') {
                prefix = "[";
                pList  = pList.substring(1);
            } else
                prefix = "";
            if (c1Match.matcher(pList.substring(0, 1)).lookingAt())
                retval[j] = prefix.length()>0 ? (prefix + c1) : c1;
            else if (c2Match.matcher(pList.substring(0, 1)).lookingAt()) {
                retval[j]   = prefix.length()>0 ? (prefix + c2) : c2;
                retval[++j] = prefix.length()>0 ? (prefix + c2) : c2;
            } else {
                assert (pList.startsWith("L"));
                retval[j] = prefix + pList.substring (0, pList.indexOf(";")+1);
                pList = pList.substring (pList.indexOf(";"));
            }
        }
        // System.out.println ("initialisation fetches:");
        // for (int kk = 0;  kk<locals && retval[kk]!=null; kk++) System.out.println
        return retval;
    }

    public static String abstractTenc (String type)  {
        if (type == null  ||  type.charAt(0) == 'L')
            return type;
        else if (c1Match.matcher(type.substring(0, 1)).lookingAt())
            return c1;
        else if (c2Match.matcher(type.substring(0, 1)).lookingAt())
            return c2;
        else if (type.charAt(0) == '[')
```

```
        return "[" + abstractTenc (type.substring(1));
    else
        assert (false);
    return null;
}

public static void   interpretInvocation (boolean isStatic, String tMethod, Stack
                                          String callerNm,  deadlockPrediction.Co
    if (! isStatic)
        typeStack.pop();
    String prtEnc = tMethod.substring(tMethod.indexOf(':')+1);
    assert (prtEnc.charAt(0) == '(');
    if (prtEnc.charAt(1) != 'V')
        for (int i = 1;  prtEnc.charAt(i) != ')';  i++)  {
            if (prtEnc.charAt(i) == 'L')  {
                typeStack.pop();          for (  ; prtEnc.charAt(i) != ';';  i++)
            } else if (c1Match.matcher(prtEnc.substring(i, i+1)).lookingAt())
                typeStack.pop();
            else if (c2Match.matcher(prtEnc.substring(i, i+1)).lookingAt())  {
                typeStack.pop();          typeStack.pop();
            }  else if (prtEnc.charAt(i) == '[') {
                if (c2Match.matcher(prtEnc.substring(i+1, i+2)).lookingAt())  {
                    typeStack.pop();          i++;
                }
            }  else
                assert (false);
        }
    String tmn = tMethod.substring (0, tMethod.indexOf(':'));
    if (tmn.charAt(0)=='\"'  &&  tmn.endsWith("\".clone"))              // point
        callerNode.noteCallTo (callerNm, "java/lang/Object.clone:()Ljava/lang/Obj
    else {
        String cn = callerNm.substring (0, callerNm.lastIndexOf("."));
        callerNode.noteCallTo (callerNm, (tmn.indexOf('.')<0 ? (cn.replace('.', '
    }
    String rt = tMethod.substring(tMethod.indexOf(')')+1).trim();
    if (rt.charAt(0) != 'V') {
        typeStack.push (abstractTenc (rt));
        if (((String)typeStack.peek()).equals(c2))    typeStack.push(c2);
    }
}

public static void  updateBBlist (int ctt, Stack tS, Object[] tA, List bcOffsets,
                                  List execStartOffset, List stackAtExecStart, Li
    if (bcOffsets.contains (new Integer(ctt))) {
        int i = 0;
        while (i < execStartOffset.size()) {
             int temp = ((Integer)execStartOffset.get(i)).intValue();
                  if (ctt >  temp)    i++;
             else if (ctt == temp)    return;
             else                     break;
```

J2SE Version	API-Specification page	New APIs count
7	http://download.oracle.com/javase/7/docs/api/	6
6	http://download.oracle.com/javase/6/docs/api/	37
1.5.0	http://download.oracle.com/javase/1.5.0/docs/api/	31
1.4.2	http://download.oracle.com/javase/1.4.2/docs/api/	59
1.3	http://download.oracle.com/javase/1.3/docs/api/	??

Table 1.3: New APIs introduced in successive J2SE Releases.

```
            }
            execStartOffset.add (i, ctt);
            stackAtExecStart.add(i, tS.clone());
            arrayAtExecStart.add(i, tA.clone());
        }
    }
    ◇
```

Fragment defined by 51, 55, 61.
Fragment referenced in 41a.

1.9 Topological Sorting of Library Code

The top-left-hand frame of the page reachable at, for instance, http://java.sun.com/javase/7/docs/api/ contains the list of 'Java-APIs' that are available with Java version 1.7. Similarly, in Table 3, pages corresponding to earlier releases are listed. Using this listing, one can discover the list of APIs released as part of the corresponding J2SE release. Taking a diff of the lists corresponding to two consequtive releases fetches the list of newly added APIs for the latter release. Accordingly we discover information listed in the column-3 of Table 3.

When implementing a newly introduced API in J2SE 7, for instance, the designer/developers can and will naturally assume and use the APIs already available in the previous version of the release, J2SE 6. In this sense the APIs would be topologically sorted, and the implementation of the new APIs introduced in a release will be depending upon (potentially) all APIs forming part of the previous release. To further clarify the notion of "API A is layered on top of API B", it suffices if, for instance, the implementation of API A instanciates objects of classes from API B, and/or invokes methods of classes therefrom.

1.10 Fragment indexes

⟨ A Short Java Program 40a ⟩ Not referenced.
⟨ Algorithmic components to transform the graphs, and analyse 18b, 19, 20b, 22 ⟩ Referenced in 6.
⟨ Attributes of the ClassInfo-object 18a, 26b ⟩ Referenced in 7b.
⟨ Attributes of the CodeBlob Instance 9, 13c, 20a ⟩ Referenced in 6.

⟨ Attributes of the lock-object 15a ⟩ Referenced in 7a.
⟨ Class declaration Processor 42 ⟩ Referenced in 41a.
⟨ ClassInfo Constructors 28a ⟩ Referenced in 7b.
⟨ ClassInfo Instance Methods 28b ⟩ Referenced in 7b.
⟨ CodeBlob Constructors 10c, 11a ⟩ Referenced in 6.
⟨ CodeBlob Instance Methods 14ab, 16 ⟩ Referenced in 6.
⟨ Collect and Report (size) statistics of the analysis ? ⟩ Referenced in 6.
⟨ Ensure node is not already created 10b, 12 ⟩ Referenced in 10c, 11a.
⟨ Globally accessable state for sharing data amongst threads 35b ⟩ Referenced in 34a.
⟨ Graph Structures comprising the state of the Analysis 10a, 11c, 13a, 17 ⟩ Referenced in 6.
⟨ JavaBlob Constructors 35a ⟩ Referenced in 34a.
⟨ JavaFile Opener 8 ⟩ Referenced in 6, 7ab.
⟨ LAG Cycle-detection and reporting 23, 24, 27f ⟩ Referenced in 7a.
⟨ Main loop to parse text-rendering and create call-graph 41b ⟩ Referenced in 41a.
⟨ Main method to execute algorithms as per the steps, to print cycles, if any 34b ⟩ Referenced in 34a.
⟨ Method Body Processors 51, 55, 61 ⟩ Referenced in 41a.
⟨ Method Header Processor 43 ⟩ Referenced in 41a.
⟨ Methods to parse text-rendering and create initial call-graph structures 41a ⟩ Referenced in 34a.
⟨ Methods to process invocation & invoke javap to create text-rendering 36ab, 38, 39 ⟩ Referenced in 34a.
⟨ OTlock Constructors ? ⟩ Referenced in 7a.
⟨ OTlock Instance Methods ? ⟩ Referenced in 7a.
⟨ Parameters, and return-type Encoder 44 ⟩ Referenced in 41a.
⟨ Print list of Containers 27e ⟩ Referenced in 26c.
⟨ Print list of Containers of classes associated with instance-locks 26c ⟩ Referenced in 27f.
⟨ Report statistics of the size of the analysis ? ⟩ Referenced in 7a.
⟨ Store cycle-signature and check to avoid repeats 26a ⟩ Referenced in 27f.
⟨ Take note of synchronized use 13b ⟩ Referenced in 11a.
⟨ The lock acquisition graph 15b ⟩ Referenced in 7a.
⟨ The text-rendering of MLtest.class 40b ⟩ Not referenced.

1.11 File indexes

"ClassInfo.java" Defined by 7b.
"CodeBlob.java" Defined by 6.
"grepNDcut" Defined by 37a.
"grepNDcutInLoop" Defined by 37b.
"JavaBlob.java" Defined by 34a.
"OTlock.java" Defined by 7a.

1.12 User-specified indexes