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}

July 31, 2015

[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: Accurate OO-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, therefore, 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. subsequently it ran a cycle-detection algorithm on the *tlog* 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 to be aquired by the `monitorenter` op-code.

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 set of utility-classes: `MethodSignature`, `TypeSignature`, and `ClassInfo`. The Java implementations for these are laid-out as below:

```
"CodeBlob.java" 6≡
    ⟨ JavaFile Opener 8c ⟩
    public abstract class CodeBlob {
        ⟨ CodeBlob Class Attributes 28a, … ⟩
```

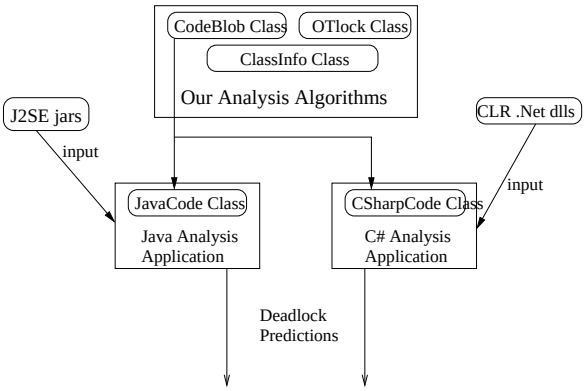

Figure 1.1: Organisation of our implementation.

⟨ *CodeBlob Instance Attributes* 26, … ⟩
⟨ *CodeBlob Constructors* 29a, … ⟩
⟨ *CodeBlob Instance Methods* 27a, … ⟩
⟨ *Algorithmic components to transform the graphs, and analyse* 31a, … ⟩
}◇

"MethodSignature.java" 7a≡
⟨ *JavaFile Opener* 8c ⟩
```
public class MethodSignature {
    String ms;
    TypeSignature[] params = null;
    public static boolean isFullySpecified (String ms) {
        return ms.substring(0, ms.indexOf(':')).indexOf('.') > 0;
    };
```
⟨ *MethodSignature Constructors* 21 ⟩
⟨ *MethodSignature component getters* 23 ⟩
}◇

"TypeSignature.java" 7b≡
⟨ *JavaFile Opener* 8c ⟩
```
public class TypeSignature {
    String type = "";
```
⟨ *TypeSignature Constructors* 24a ⟩
⟨ *TypeSignature Static Initialisation* 24b ⟩
⟨ *TypeSignature Methods* 25 ⟩
}◇

"OTlock.java" 8a≡
⟨ *JavaFile Opener* 8c ⟩
```
public class OTlock {
```
⟨ *The Type lock order graph* 38a ⟩
⟨ *OTlock Instance Attributes* 38b ⟩
⟨ *TLOG Cycle-detection and reporting* 57, … ⟩
```
}◇
```

"ClassInfo.java" 8b≡
⟨ *JavaFile Opener* 8c ⟩
```
public class ClassInfo {
```
⟨ *ClassInfo Instance Attributes* 52a, … ⟩
⟨ *ClassInfo Constructors* 63a ⟩
⟨ *ClassInfo Instance Methods* 63b ⟩
```
}◇
```

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* 8c ⟩ ≡
```
/* file name -- Lock Acquisition Graph Construction, Cycle detection / reporting
 *
 * author: Vivek K. Shanbhag, IIIT-B, Bangalore, India.
 */

package deadlockPrediction;

import java.util.*;
import java.util.concurrent.*;
import java.io.*;
◇
```
Fragment referenced in 6, 7ab, 8ab.

1.2.4 Phase 3: Accurate OO-analysis of OO-Libraries

There are a few observations we need to make before proceeding further with the discussion:

- In procedural analysis of imperative languages, the edges of the call graph are discovered through using the compile-time types of objects. The compile-time types of objects in the program are the same as their declared-types. Whereas, in the case of OO-Languages like C++/Java/C#, method-invocations on objects are resolved based upon their run-time types. Their run-time types are determined at the time of object-instantiation.

- Given this "dynamic-dispatch" feature of OO-Languages, it becomes possible for a given method-implementation of some type to demonstrate different "perceived-behaviours" depending upon the run-time types of the arguments passed to its different invocations.

- A prime motivation for static-time analysis algorithms has been that the size of the analysis be implied by the size of the program, rather than the size of the data-range that the program can operate upon. In the good-old days of Pascal/C Programming, the basic types provided by the Language were few, and there were just two language-constructs that allowed creating 'higher-level' types: *struct*/*typedef* and *union*. Back then, the 'type-system' part of the language constructs had very little implication on the interpretation of the behaviour of a program (over and above) as expressed through using the rest of the language constructs. So when one tries static-time analysis of a C-Language program, the interpretation of the behaviour of the program, the size of (the expression of the) program, and the size of its analysis can roughly be seen to be in direct proportion to each other.

- Whereas, with OO-notations like Java/C++/C#, these things have changed. Comparitively, these notations have a much rich-er component that is intended for use to create 'higher-order' types, and very intersting, and complex relations between them. Using such features of these notations, it is now possible to express program behaviour in a more compact form. Inheritance (in Java, for instance) is a very good example: all sub-classes of *java.lang.Object* inherit all the public-functionality programmed for it by the library. Thereby, every object in the application is able to (potentially) demonstrate the behaviour it so inherits at virtually zero-cost, as reflected in the size of the (expression) of the application-program. Such notational convinience can be used very effectively to create relatively brief specifications of program behaviour.

- Consequently, notice two implications:

  - When static-time analysis is conducted on such OO-notation, without giving due respect to the implications of its underlying type-system, inaccuracy is bound to creep into its results. In our Phase 2 attempt, where we

conducted procedural-analysis with no regard to the 'virtual function mechanish', our analysis was indeed inaccurate, but the size of our analysis was (roughly speaking) in direct proportion to the size of the (expression of the) program.

– Now that we want to conduct a more accurate OO-analysis of the OO-notation, we need to *un-fold* the expression of the program with due respect to its under-lying type-system, to un-cover the complete programmer-intended interpretation of the behaviour of the program. Our analysis therefore needs to create a representation of the program that corrects for the notational convinience provided in the form of the type-system constructs. Thus, while the size of our analysis is still in direct proportion to the program-size, the constants are larger than that for an imprecise analysis.

- More specifically, consider the following example:

"Observer.java" 10≡

```
interface DoSomethingBadToday {    public void doIt();  };

class GoDivideByZero implements DoSomethingBadToday {
    public void doIt () {        int retval = 55/0;    };
};

class GetIntoInfiniteLoop implements DoSomethingBadToday {
    public void doIt () {        for ( ; ; );          };
};

public class Observer {
    public static void howAboutToday (DoSomethingBadToday howTo) {
        howTo.doIt();
    };

    public static void main (String[] args) {
        DoSomethingBadToday x = null;
        howAboutToday (x = new GetIntoInfiniteLoop());
        howAboutToday (x = new GoDivideByZero());
    };
};
```

◇

- Procedural-analysis of the above code computes a single (imprecise) model for the bahaviour of *Observer.howAboutToday*: that based upon the compile-time (declared) type of its *argument*, namely, *DoSomethingBadToday*. In fact, this model is not *intend-able* at any calling point since the type: *DoSomethingBadToday* is an abstract type, and there can be no object whose run-time (instantiation) type is abstract.

- Whereas, in reality, the expression of the specification of the said method actually contains (at-least) two potential interpretations: one where the run-time type of its *argument* is *GetIntoInfiniteLoop*, and another when it is *GoDivideByZero*. We would like our more accurate analysis to be able to infer which of these is intended, based upon the calling-context, at the point of invocation.

- Executing the command: "`javac Observer.java; javap -private -c -verbose Obsever`" produces a text-rendering of the compiler-generated class-file, that contains the following snippet:

```
...
public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=2, Args_size=1
   0: aconst_null
   1: astore_1
   2: new #3; //class GetIntoInfiniteLoop
   5: dup
   6: invokespecial #4; //Method GetIntoInfiniteLoop."<init>":()V
   9: dup
   10: astore_1
   11: invokestatic #5; //Method howAboutToday:(LDoSomethingBadToday;)V
   14: new #6; //class GoDivideByZero
   17: dup
   18: invokespecial #7; //Method GoDivideByZero."<init>":()V
   21: dup
   22: astore_1
   23: invokestatic #5; //Method howAboutToday:(LDoSomethingBadToday;)V
   26: return
  LineNumberTable:
   line 17: 0
   line 18: 2
   line 19: 14
   line 20: 26

...
```

- More specifically, (since our more accurate analysis is built by modifying our initial procedural-analysis of the code-base), in the above example we would like our analysis to do the following:

  1. Upon encountering the *definition* of the `Observer.howAboutToday()` method, our analysis creates a *code-blob* with its *compile-time* method-signature as the node-name. It then conducts a `CompileTimeType`-level analysis of the associated method-body, and installs *caller-called* links between this node and the nodes corresponding to all the called mathods.
  2. Subsequently, (if found necessary), while conducting `RunTimeType`-level analysis of `Observer.main`, on encountering the *invocation* of `Observer.howAboutToday`, on byte-code index 11:, above, our OO-analysis must be able to firstly tell that although the declared (compile-time) type of variable *x* is *DoSomethingBadTaday*, that its (instantiated) run-time type is actually *GetIntoInfiniteLoop*, and secondly, *re-interpret* the method-body of `Observer.howAboutToday()`,

given the info that the actual (run-time) type of the argument is different from its compile-time parameter-type. The new behaviour fetched from such abstract-interpretation is then saved under a fresh *code-blob* whose name is the method-signature dereived from the run-time types involved in the invocation.

3. Similarly, later, on encountering another invocation on byte-code index 23:, above, our OO-analysis must be able to firstly tell that although the declared (compile-time) type of variable *x* is *DoSomethingBadTaday*, that its (instantiated) run-time type is actually *GoDivideByZero*, and secondly, *re-interpret* the method-body of Observer.howAboutToday(), given the info that the actual (run-time) type of the argument is different from its compile-time parameter. The new behaviour fetched from such abstract-interpretation is then saved under a fresh *code-blob* whose name is the method-signature dereived from the run-time-types involved in this second invocation.

- Notice from the above elaboration, that although the input source contains merely one specification of the behaviour of the method called `Observer.howAboutToday()`, our OO-analysis creates as many distinct models of its behaviour as can manifest owing to the run-time method-binding feature of the language.

- Re-stating differently, therefore, our analysis views method-bodies (as available in the input-source) as being template-specifications for method-behaviour, and it instantiates as many *varients* of the template as can manifest, given the different combinations of run-time-types for input-parameters. All these instantiated varients are stored, and re-used wherever the actual-parameter-type-lists match precisely.

- In the example above we have seen one source of variability in the interpretation of the specification of a method behaviour, namely the one that is sourced from the possibility of invoking a method by passing parameters of run-time types that are sub-classed from those specified as the (compile-time) declared types of the corresponding arguments. There is another source of variability in the interpretation of the specification of a method-behaviour: one which is fetched from the possibility of invoking an inherited method on an object of a type sub-classed from the one that defines the method-behaviour.

- Irrespective of the source of variability of interpretation, the way to proceed to fetch the *Code-Blob* that will represent the referenced behaviour is to take the byte-code representing the specification of the method-behaviour, and initialise its local-variable-types array based upon the run-time-types of target-object, and the parameters passed to the invocation, and then proceed with the abstract-interpretation of the byte-code.

- Consider another example, below:

`"B.java"` 12≡

```
import java.util.*;
interface X {    public void xx (java.util.Collection c);  };

class A implements X {
    public void xx (java.util.Collection c) {
        c.add ("I");  c.add ("am");
        String here = "here";
        c.add (here); c.add (here);
        System.out.println ("Cardinality: " + c.size());
        java.lang.Object[] a = c.toArray();
        System.out.println ("Finally: " + a[a.length-1].toString());
    };
};

public class B extends A {
    public void xx (java.util.SortedSet ss) {
        ss.add ("Where");   ss.add ("are");
        String u = "you";   ss.add (u);         ss.add (u);
        System.out.println ("Cardinality: " + ss.size());
        java.lang.Object[] a = ss.toArray();
        System.out.println ("Finally: " + a[a.length-1].toString());
    };

    public static void main (String[] args) {
        java.util.SortedSet ss = new TreeSet ();
        X odXiB = new B ();    odXiB.xx (ss);     ((B)odXiB).xx (ss);
    };
};
◇
```

In the code (`B.java`), above, Class `B` is a public subclass of Class `A` which implements interface `X`. `A` is a concrete class that implements the method `xx` as required by interface `X.xx(...)`. Consequently, our analysis will create a CodeBlob for `A::xx(Collection)V`. The name `X` will get inserted into the `interfaceNames` attribute of the `CodeBlob` class. The command "`javac B; javap -private -c -verbose`" generates the text-rendering of the byte-code corresponding to 'B.class', that contains the following sippet:

```
public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=3, Args_size=1
   0: new #18; //class java/util/TreeSet
   3: dup
   4: invokespecial #19; //Method java/util/TreeSet."<init>":()V
   7: astore_1
   8: new #20; //class B
   11: dup
```

```
 12: invokespecial #21; //Method "<init>":()V
 15: astore_2
 16: aload_2
 17: aload_1
 18: invokeinterface #22,  2; //InterfaceMethod X.xx:(Ljava/util/Collection;)V
 23: aload_2
 24: checkcast #20; //class B
 27: aload_1
 28: invokevirtual #23; //Method xx:(Ljava/util/SortedSet;)V
 31: return
LineNumberTable:
 line 25: 0
 line 26: 8
 line 27: 31
```

It is particularly interesting to discuss the two invocations of the method `xx` at byte-code indices 18, and 28, in the above snippet: Both these invocations are for the target object of run-time-type `B`, and the run-time-type of the argument passed to both invocations is an object of type `java.util.SortedSet`. However, the actual method-bodies that are handed-control in both the invocations are distinct. This is so because the compiler also uses information about the type of the reference/handle to the object when resolving calls. So, as learnt from the `Observer` example, if we try to create `CodeBlob` objects named *only* on the basis of the RTTI of the target-object, and the method-arguments, then we will end up with a name-clash (when inserting them into the `allNodes` collection) for this example-case.

- Consider yet another example, below, before we proceed to design the data-structures to help conviniently implement our OO-analysis.

"V.java" 14≡

```
import java.util.*;
class U {
    public void m () {  System.out.println ("Invoked U::m()V;");  };

    public static void staticMethod (U u, Boolean x)  {
        System.out.println ("Invoked: " + u.getClass().getName() + ", " +
                            (x ? "true" : "false") + ".");
        U uuu = x ? new U() : new V();  uuu.m();
        U uu = null;
        if (x)  {  uu = new U();         uu.m(); }
        else    {  uu = new V();         uu.m(); }
                                          u.m();
    };
};

public class V extends U {
    public void m () {  System.out.println ("Invoked V::m()V;");  };
```

```
        public static void main (String[] args) {
            U.staticMethod (new U(), true);
            U.staticMethod (new U(), false);
            U.staticMethod (new V(), true);
            U.staticMethod (new V(), false);
        };
    };
    ◇
```

- Let us look at the following example:

"Hmp.java" 15a≡

```
    import java.util.concurrent.*;
    import java.util.*;
    class Hmp {
        public static void main (String[] args) {
            ConcurrentHashMap hmp = new ConcurrentHashMap ();
            hmp.put ("one", 1);
            hmp.put ("one", 11);
            System.out.println ("hmp.size() returns " + hmp.size());
        };
    };
    ◇
```

"N.java" 15b≡

```
    import java.util.*;
    class M {
        public  int  i    =  15;
        public  void m ()    {  System.out.println ("Invoked M::m()V; Mi = " + i);  }
        private void mpri () {  System.out.println ("Invoked M::mpri()V;");  };
                void mpub () {  System.out.println ("Invoked M::mpub()V;");  };
    };

    public class N extends M {
        public  int  i    =  20;
        void mpri () {  System.out.println ("Invoked N::mpri()V; Ni = " + i);  };
        void mpub () {  System.out.println ("Invoked N::mpub()V; Ni = " + i);  };


        public static void main (String[] args) {
            M om = new M(); om.m();
            //System.out.println ("om.i = " + om.i + " : " + ((N)om).i);
```

```
          om = new N(); om.m();
        System.out.println ("om.i = " + om.i + " : " + ((N)om).i);
    };
};
◇
```

Data-Structure Design

In view of the above listed observations, we can now proceed to try evolve the design of the data-structure (called the `CodeBlob`), that will be the node of the call-graph. An `CodeBlob` object is used to store the analysis of either an entire method-body, or just a *synchronized* block of statements. The analysis of a `CodeBlob` node can be conducted at one of three levels of accuracy: `Null`, `CompileTimeType`, or `RunTimeType`. The essential attributes of the `CodeBlob` are the method-signature that it represents. There are three sources, wherefrom we infer method-signatures:

1. The first source is when we encounter member-method-definitions as we parse the text-rendering of the input-jars. Recollect that a `jar` is a collection of `.class` files; the `javap` tool renders the contents of these individual `.class` files into a sequence of their corresponding text-rendering; such individual text-rendering starts with a class-header followed by a *open-brace*, followed by the sequence of member-method-definitions, and finally signalled by the occurrance of a matching *close-brace*.

2. The second source is the significant comment on the same line as the `invoke` family of byte-codes (within the text-rendering of the member-method-body-definition) therein. This is encountered, and taken-note-of, when conducting `CompileTimeType`-level analysis of the method body of the member-method-body-definition. It is important to mention here, that method-signatures fetched from this source may not be fully-qualified. In particular, when the invoking-method and the invoked method belong to the same class as the containing class, it is possible that the method-signature is not prefixed with the fully-qualified class-name of the containing-class.

   Never-the-less, when it comes to storing a node for the invoked-method, the key is indeed the fully-qualified method-signature; Of which the post-fix is taken from the said significant-comment, and the prefix, if-and-where-necessary, is generated using the fully-qualified-name of the containing class.

3. The third source is the reconstructed method-signature, (when conducting the `RunTimeType`-analysis of the method-body), using actual types of the target-object, and the invocation-arguments for the said method, when doing abstract-interpretation of the called-method-body, given the types handed from the calling-context.

- Of these, method-signatures from the first source are seen along with the definition of their method-body (if it is available to the analysis at all). Specifically, for instance, methods that are explicitly defined to be either native, abstract, or interface methods, will not have their behaviours defined/available for analysis. Signatures fetched from the second source, namely the significant comments, encapsulate all the compile-time-type-level analysis and method resolution related computation conducted by the compiler. The third set of method-signatures are the outcome of the abstract-interpretation conducted by our OO-analysis, using information available in the form of the significant comment, and the class-heirarchy-tree to identify the method-body intended to be invoked, and to discover the interpretational-varient of the method-body given the run-time-type information available from the calling-context.

- Method-signatures with method-body-definitions fetched from the first two sources are stored in a `definedCode` class-attribute; those among them that do-not have method-bodies defined are stored in the list of either (native methods) `notDefinedCode`, or (abtract / interface methods) `tobeDefinedCode`.

- `definedCode` is a Map: from the method-signature to the `CodeBlob`-object. `notDefinedCode` is merely a Collection of method-signatures. `tobeDefinedCode` is Map: from method-signatures to a Collection of `CodeBlob`-objects. All of these are class-attributes of the `CodeBlob` class. The members of the `notDefinedCode` Collection, and the *key*s from the other two *Map*s are all method-signatures that are fetched from the first and/or second source, discussed above.

- Another class-attribute (`varients`) of the `CodeBlob` class stores all the interpretational-varients in the form of a Map: from a method-signature based key to the `CodeBlob`-object that represents the said varient. The "key-value" for this map has the form: *rtms-=varientOf=-ctms*; where *rtms* stands for the Run-Time-Method-Signature generated from source-3, and *ctms* stands for the Compile-Time-Method-Signature generated from source-2. Such `CodeBlob`s, from the `varients` map can get referenced from the `calledCode`, and `callingCode` instance-attributes of nodes on the `definedCode` Map, but these (varient) objects themselves won't get onto the `definedCode` Map.

- In Step-1, where all the input is read once, all `CodeBlob`-objects fetched from the first source are unconditionally *analysed* at the `CompileTimeType`-level. Method-signatures fetched form the second-source are *analysed* at the `Null`-level. During this phase, since nodes are not yet analysed at the `RunTimeType`-level, there is no need to generate nodes or method-signatures from the third-source.

- in Step 2 we look through the `definedCode` Map and single out all the `Null`-analysed nodes to identify their parent classes (if any) where-from they might be able to inharit any behaviour. If such a method is found, then its behaviour-specification is re-interpreted to populate the `calledCode` attribute of the inheritor method.

- Handles on all *synchronized*-nodes are additionally recorded in the `sCode` attribute. In Step 3, we parse the back-edges (`callingCode`) from all `sCode`-nodes to discover all their public caller-methods. These we collect into an (instance-attribute) set called `startPoints`.

- In Step 4, (after the value for `CodeBlob::startPoints` is computed in Step-3), the Tlog-edge detection starts from the various 'startPoints', to discover connectivity among nodes of the TLOG. When doing so, every encountered node, enroute, is firstly upgraded to `RunTimeType`-level, if necessary. Analysis of behaviour at the `RunTimeType`-level generates new behavioural-varients; some of which could be `synchronized`, and whose `startPoints` attributes are not yet populated. This method behaviour must therefore compute the value for this attribute for such newly created nodes, and implement the Step-4 for them as well.

- Steps 3, and 4, therefore could also be done in a loop, until the fixed-point is reached, wherein no more `sCode`-nodes are creared when executing Step-4 of the loop-iteration.

- We now need to discuss the basis to decide to create a new node when conducting `RunTimeType`-Analysis. Nodes have already been created in Step-1, corresponding to method-signatures sourced from sources 1 & 2. When conducting RTTA (in Step-4), we have discovered the *rms*(Run-time method-signature), and the *cms*(compiler-generated method-signature) corresponding to the method-invocation. If the *cms* matches something in the `definedCode`, and if it has its `accuracy` set to `CompileTimeType`-Analysis, we are proceed as per subcase-1; else if the matched node from `definedCode` is `Null`-analysed, then we proceed as per subcase-2. In case that the search for *cms* in `definedCode` fetches a blank, then one needs to further walk-up the inheritance tree of the *rcms* (as further-explained in the next point), to identify if we can find a *bahaviour-specification* for the method that might bind to this call. If such a `behaviourSpec` is found, then we proceed similar to subcase-1, or if not, then we assert-failure!

- *rcms* stands for run-time-prefixed compiler-generated-postfixed method-signature. This is fetched very easily, given the *cms* of the invoked method, and the state of the operant-type-stack, as maintained by the `RunTimeType`-Analysis. As the abbrevation states, we use the type-name of the object that is the target of the method-invocation (using it as the prefix), and augment to it the un-qualified method-name followed by the parenthesized encoding of the parameter-list and terminated by the return-type-encoding (using this entire string as the post-fix), as-from the *cms*.

- Subcase-1 proceeds as follows: The node identified from the `definedCode` list provides the *behavioural-specification* for the varient behaviour as (to be) interpreted for the *rms*. This node whose *rms* is distinct from the *cms* is stored in the Map `varients`, and the key for its entry in the Map `calledCode` is *rms-:compiledAs:-cms-=varientOf=-dms*. The node identfied from the `definedCode` Map provides the *dms: defined* method-signature.

- Subcase-2 corresponds to the invocation of some method, the behaviour-specification of which has not been made available to the analysis, so we merely proceed to make an entry for such a call in the `calledCode`, using the entry: $<$ *rms-:compiledAs:-cms*, null $>$.

- There is an interesting space & time optimisation that can be employed: that of `Null`-analysing `code-blob`s that have an abstract- or interface- type as either the target object-type, or as one of their formal-parameter-types. This be implemented. There is an interesting space & time saving approximation: that of choosing to not re-interpret a calling-context where-from a called-blob returns an actual-type that is a sub-class of its declared return-type. The cost of not-implementing this approximation must be reported.

- The implementation explores an on-the-fly implementation in the interest of minimising the use/need for time and space required for the execution, given that the size of the analysis is all set to blow up significantly, otherwise. We analyse individual code-blobs at three levels of precision depending upon whether or not they become known to be potentially participating in some tlog-cycle.

- Our analysis requires only a single pass through the text-rendering of the input jars. In this code scan, the first pass through the method-body causes its contents to be unconditionally stored in a `code` array. While doing this, we instatiate the caller-called edges of the call-graph based upon the compile-time type information present in the input. This infomation may be imprecise, but it is better than nothing. The called `CodeBlob`s created during this pass are named based upon the compile-time types.

- For such nodes among these, which are either found to be *synchronized*, or making the use of the `monitor-enter-exit` byte-codes, that qualify to be inserted into the `sCode` list, we invoke the abstract-interpretation pass. This discovers the run-time type-information for objects / invocations within the method-body, and instantiates caller-called edges based upon this more precise RTTI.

- At the end of the input-source, we are left with some nodes that are compile-time-type analysed, and others that are run-time type-analysed. Moreover, these can have edges pointing to `Null`-analysed nodes that are interpretational varients of `definedCode`-nodes. Such need to be run-time type-analysed. However, this is done only for those nodes encountered in the path of cycle-detection, on the fly.

- Further, we implement an additional step: first collect the names of all ”public” methods that *can* invoke any method from the `sCode` list, into a collection called the `startPoints`. Second: starting from every call-path rooted at every member of the `startPoints` set, look for cycles while ensuring that the analysis level of every method-body encountered on the path is conducted at the run-time-type-analysis level.

- The above is all the computation we need to do, really. We cache partial results but compute it only for code-blobs that are relevant to finding cycles. For all others that we notice, we do not spend any time, nor use-up any space exploring.

The design of the data-structure for the `CodeBlob` class, as explored in descriptive english text above, is more-formally specified in the form of various program invarients listed below:

$$Class - Attribute\,'CodeBlob :: definedCode' \equiv \{n_i = \langle\ ms_i \rightarrow mb_i\ \rangle\}, where$$
$$method - signature\ ms_i\ is\ fetched\ from\ source - 1,\ or\ source - 2,\ and$$
$$mb_i\ \neq\ null,\ and\ mb_i.nn\ =\ ms_i. \quad (1.1)$$
$$Class - Attribute\,'CodeBlob :: notDefinedCode' \equiv \{ms_i\}, where$$
$$method - signature\ ms_i\ fetched\ from\ source - 1\ defines\ a\ native\ method. \quad (1.2)$$
$$Class - Attribute\,'CodeBlob :: tobeDefinedCode' \equiv \{\langle\ ms_i \rightarrow VCB_i\ \rangle\}, where$$
$$method - signature\ ms_i\ fetched\ from\ source - 2\ defines\ an\ abstract\ or\ interface\ method,\ and\ the$$
$$Map\ VCB_i = \{\langle rms_1 \rightarrow vcb_i \rangle\} :\ a\ set\ of\ RTTI\ CodeBlobs,\ whose\ method - signature\ rms_i\ is$$
$$fetched\ from\ source - 3,\ through\ abstract - interpretation\ of\ method - invocation\ which\ the$$
$$compiler\ resolved\ against\ method - signature\ ms_i,\ and\ vcb_i\ is\ its\ varient\ code\ behaviour. \quad (1.3)$$
$$Instance - Attribute\,'cb.analysis'\ is\ initialised\ to$$
$$'Null'\ if\ cb.nn\ was\ fetched\ from\ source - 2\ or\ source - 3,$$
$$'CompileTimeType'\ if\ cb.nn\ was\ fetched\ from\ source - 1,\ and\ not\ synchronized,$$
$$'RunTimeType'\ if\ cb.nn\ was\ fetched\ from\ source - 1,\ and\ is\ declared\ synchronized. \quad (1.4)$$
$$Instance - Attribute\,'cb.calledCode' \equiv \{\langle\ ms_i \rightarrow mb_i\ \rangle\},\ where$$
$$(cb.analysis = Null) \rightarrow\ (i\ =\ 0) \wedge$$
$$((cb.analysis \neq Null) \wedge (mb_i \neq null) \wedge (ms_i = mb_i.nn)) \rightarrow$$
$$(CodeBlob :: definedCode[ms_i] = mb_i) \wedge$$
$$((cb.analysis = CompileTimeType) \wedge (mb_i = null)) \rightarrow$$
$$((ms_i \notin CodeBlob :: definedCode.keySet()) \wedge (ms_i \in CodeBlob :: tobeDefinedCode.keySet())) \wedge$$
$$((cb.analysis = RunTimeType) \wedge (ms_i \neq mb_i.nn) \wedge (ms_i \in CodeBlob :: definedCode.keySet())) \rightarrow$$
$$((mb_i \in CodeBlob.definedCode[ms_i].varients) \wedge (mb_i.behaviourSpec = ms_i))$$
$$((cb.analysis = RunTimeType) \wedge (ms_i \neq mb_i.nn) \wedge (ms_i \notin CodeBlob :: definedCode.keySet())) \rightarrow$$
$$((mb_i \in CodeBlob.tobeDefinedCode[ms_i].keySet()) \wedge (mb_i.behaviourSpec = ms_i)). \quad (1.5)$$
$$cb.calledCode.keySet()\ contains\ method - signatures\ fetched\ only\ from\ source - 1,\ or\ source - 2. \quad (1.6)$$
$$Instance - Attribute\,'cb.callingCode' \equiv \{\langle\ ms_i \rightarrow mb_i\ \rangle\},\ where$$
$$((cb.analysis = Null)! \rightarrow (i\ =\ 0) \wedge (mb_i \neq null) \wedge (ms_i = mb_i.nn)) \quad (1.7)$$
$$cb.callingCode.keySet()\ can\ contain\ method - signatures\ fetched\ from\ sources - 1\ /\ -2,\ or\ -3. \quad (1.8)$$
$$Instance - Attribute' cb.varients' \equiv \{\langle rms_i \rightarrow mbv_i \rangle\}, where$$
$$rms_i\ are\ method - signatures\ fetched\ from\ source - 3,\ and, (rms_i = mbv_i.nn), and$$

mbv_i *is the corresponding* $CodeBlob$ *object that cannot store into the* $definedCode\ Map$. (1.9)

1.2.5 The call-graph, its transformations, and the TLOG

We start our analysis from a cleanly built J2SE Workspace, and proceed as detailed, below in this section, to build the call-graph to represent the input, and concurrently, to build its Type-level Lock Order Graph. We build upon the data-structure described above, and implement operations on them alongside discusion of the analysis. We use an illustrative running example to further facilitate clarity and illustrate 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(); };
}
```

Preamble: The 'Signature' utility classes

We collect all the code relating to encoding/decoding of the Method, and Type signatures into these two classes, so as to minimise opportunity for avoidable String-handling code-errors. Moreover, we can use the Java-scheme for encoding of the signatures for now, and make any modifications if necessary for porting the C++-analysis, on top.

⟨ *MethodSignature Constructors* 21 ⟩ ≡

```
public MethodSignature (String ms) {
    assert (ms.indexOf (' ') < 0  &&  ms.indexOf (':') > 0);
    assert (ms.indexOf ('(') > 0  &&  ms.indexOf (')') > 0);
    this.ms = ms.intern();
    if (fsmName().contains("/"))
       this.ms = fsmName().replace ('/', '.') + ":" + prtEncoding();
    if (ms.indexOf("()") < 0)
        decodeParamEncoding ();
    //String clsNm = fsmName();  clsNm = clsNm.substring (0, clsNm.lastIndexOf('.'))
    //if (clsNm.startsWith("\"")  &&  clsNm.endsWith("\"")) {
    //    clsNm = clsNm.substring (1, clsNm.length()-1);
    //    this.ms = clsNm + dropClassNmGetRest();
    //};
};
```

```
public MethodSignature (String cN, String ms) {
    String fsmn = ms.substring (0, ms.indexOf(':')).replace('/', '.');
    // assert (! fsmn.startsWith(cN));
    this.ms =  fsmn.indexOf('.') < 0  ?  (cN + "." + ms)
                                      :  fsmn + ms.substring (ms.indexOf(':'));
    //this ((this.ms = ms.substring (0, ms.indexOf(':')).replace('/', '.')).indexOf(
    //      ? (cN + "." + ms) : this.ms + ms.substring (ms.indexOf(':')));
    if (ms.indexOf("()") < 0)
        decodeParamEncoding ();
    //String clsNm = fsmName();  clsNm = clsNm.substring (0, clsNm.lastIndexOf('.'))
    //if (clsNm.startsWith("\"")  &&  clsNm.endsWith("\"")) {
    //    clsNm = clsNm.substring (1, clsNm.length()-1);
    //    this.ms = clsNm + dropClassNmGetRest();
    //};
};

public MethodSignature (String cN, String mN, String pp, TypeSignature rt) {
    pp = pp.substring(1, pp.length()-1);
    String ppp = "";
    for (int i = 0, j = 0;  i < pp.length();  i++)  {
        if (pp.charAt(i) == '<')           j++;
        if (j == 0)                        ppp += pp.charAt(i);
        if (j>0  &&  pp.charAt(i) == '>')  j--;
    }
    String[] pTypes = ppp.split(",");
    ms = cN + '.' + mN + ":(";
    if (pp.equals(""))
        params = null;
    else {
        params = new TypeSignature [pTypes.length];
        for (int i = 0;  i < pTypes.length;  i++) {
            //if (pTypes[i].trim().endsWith(">"))
            //    pTypes[i] = pTypes[i].substring (0, pTypes[i].indexOf("<"));
            params[i] = new TypeSignature (pTypes[i].trim(), false);
            ms += params[i].signature();
        }
    }
    ms += ")" + rt.signature();
};

private void decodeParamEncoding () {
    String pp = parenParams();
    assert (pp.startsWith ("(")  &&  pp.endsWith (")"));
    LinkedList l = new LinkedList();
    int rv = 0; String temp = "";
    for (int i = 1;  i < pp.length()-1;  rv++) {// post-incr i when consumed.
        while (pp.charAt(i) == '[') {
            temp += '[';  i++;
        }
        if ("BCFISZDJ".indexOf(pp.charAt(i)) >= 0) {
```

```
                l.add (rv, temp + pp.charAt (i++));
                temp = "";
            } else if (pp.charAt(i) == 'L') {
                // System.out.println ("pp(i...):" + pp.substring(i));
                l.add (rv, temp + pp.substring (i, pp.indexOf(';', i) + 1));
                i = pp.indexOf (';', i) + 1;
                temp = "";
            } else {
                System.out.println ("Offending type-symbol: " + pp + ":"+i+ "?");
                assert (false);
            }
        }
        if (rv > 0) {
            params = new TypeSignature [rv];
            for (int i = 0;  i < rv;  i++)
                params[i] = new TypeSignature ((String)l.get(i), true);
        }
        l.clear();
    };
    ◇
```

Fragment referenced in 7a.

⟨ *MethodSignature component getters* 23 ⟩ ≡

```
    public TypeSignature className() {
        return new TypeSignature (ms.substring (0, fsmName().lastIndexOf('.')), false);
    };

    public String dropClassNmGetRest () {
        return ms.substring (fsmName().lastIndexOf('.')+1);
    };

    public String signature ()   { return ms;                                  };
    public String fsmName ()     { return ms.substring (0, ms.indexOf(':'));  };
    public String prtEncoding () { return ms.substring (ms.indexOf(':') + 1); };
    public String methodName ()  {
        return fsmName().substring (ms.lastIndexOf('.') + 1);
    };

    public TypeSignature retType () {
        return new TypeSignature (ms.substring (ms.lastIndexOf (')') + 1), true);
    };

    public String parenParams () {
        return ms.substring (ms.indexOf(':')+1, ms.lastIndexOf(')')+1);
    };

    public int paramCount ()     { return (params == null  ?  0  :  params.length);  };
```

```
    public TypeSignature[] paramList () { return params;  };
    ◇
```

Fragment referenced in 7a.

⟨ *TypeSignature Constructors* 24a ⟩ ≡

```
    public TypeSignature (String type, boolean decode) {
        if (type == null)  type = "";
        if (decode) {
            for (this.type = "";  type.startsWith("[");  type = type.substring(1))
                this.type += "[]";
            if      ("B".equals(type))   this.type = "byte"    + this.type;
            else if ("C".equals(type))   this.type = "char"    + this.type;
            else if ("D".equals(type))   this.type = "double"  + this.type;
            else if ("F".equals(type))   this.type = "float"   + this.type;
            else if ("I".equals(type))   this.type = "int"     + this.type;
            else if ("J".equals(type))   this.type = "long"    + this.type;
            else if ("S".equals(type))   this.type = "short"   + this.type;
            else if ("Z".equals(type))   this.type = "boolean" + this.type;
            else if ("V".equals(type)  ||
                     "".equals(type))    this.type = "void"    + this.type;
            else  {
                if (type.startsWith("L")  &&  type.endsWith(";"))
                    this.type = type.substring(1, type.length()-1).replace ('/', '.') + t
                else if (type.startsWith("c1") ||  type.startsWith("c2"))
                    this.type = type + this.type;
                else
                    assert (false);
                }
        } else
            this.type = type.equals("") ? "void" : type;
        if (this.type.startsWith("L") &&  this.type.endsWith(";")) {
            System.out.println("Offending Type in: "+ type);
            assert (false);
        }
    };
    ◇
```

Fragment referenced in 7b.

⟨ *TypeSignature Static Initialisation* 24b ⟩ ≡

```
    static Map basicTypes = new HashMap ();
        static {
            basicTypes.put ("byte",    "B");
            basicTypes.put ("char",    "C");
            basicTypes.put ("double",  "D");
            basicTypes.put ("float",   "F");
```

```
            basicTypes.put ("int",     "I");
            basicTypes.put ("long",    "J");
            basicTypes.put ("short",   "S");
            basicTypes.put ("void",    "V");
            basicTypes.put ("",        "V");
            basicTypes.put ("boolean", "Z");
        };
        public static String basicEncoded = "BCDFIJSVZ";
    ◇
```

Fragment referenced in 7b.

⟨ *TypeSignature Methods* 25 ⟩ ≡

```
    public String signature () {
        String rv = "", t = type;
        while (t.length() > 0  &&  t.endsWith("[]"))  {
            rv += "[";
            t = t.substring (0, t.length()-2);
        }
        if (basicTypes.keySet().contains (t))
            rv += (String) basicTypes.get (t);
        else if (t.length()>0)
            rv += "L" + t.replace ('.', '/') + ";";
        return rv;
    };
    public String name () { return type; };
    ◇
```

Fragment referenced in 7b.

Step 0: Specifying input

The input to our analysis 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.h (detailed) below. Our analysis scans through the source only once, storing all the method-bodies so that they can be re-scanned later, if necessary, without having to read the input.

Step 1.a: Essential instance-attributes of a node of the call-graph

Corresponding to every method, either, whose method-body is parsed, or whose method-invocation is noticed (while parsing the method-body of some other invoking method), we create a node to represent it. The collection of such nodes that represents the entire input to our analysis is stored in a keyed map (`definedCode`) for which we use the two-tuple $<$ fully-qualified-method-name-as-key, node-to-represent-it $>$ as the way to store this information. The data-attributes that we record about the method-body are as under:

⟨ *CodeBlob Instance Attributes* 26 ⟩ ≡

```
private String  lockType = null;
    // Stores the type of object that is synchronised upon:
    //  Null implies an 'un-Synchronised' code-blob
    //  ".class" postfix for static synch methods
    //  setLockType() ensures consistancy with the locks hashMap.
public boolean isPublic = false;    // Is this a public method ?
public boolean isStatic = false;    // Is this a static method ?
//private String  nodeName      = null; // Name of the node
private MethodSignature nodeName= null;
    //  String "package-name/class-name.method-name:prtEncoding"
//private String  containerType = null; // Type of the container Class
//private String  returnType    = null;
//private int     noOfParams    = -1;
//private String[] paramList    = null;
```

◇

Fragment defined by 26, 27b, 28bc, 36a, 46b, 52b.
Fragment referenced in 6.

When reading the description below, kindly, bear in mind that the `CodeBlob` object can be used to represent either the method body, as is seen in the input, or also any of its various interpretational-varients, as manifest during run-time depending upon the actual typess of the target-object, and the invocation arguments. The `lockType` attribute for an un-*synchronized* method continues to remain null. `containerType` stores the fully-qualified name, either, of the defining-class of the method (in case this node corresponds to a method-body whose definition is seen in the input), or of the run-time-type of the object that is the target for the method-invocation. Similarly, the `returnType` stores either the declared compile-time return-type as seen from the input method-definition, or the lowest-common type in the class-heirarchy that is the common parent of all the RTTI-types corresponding to the various reachable return-statements in the byte-code corresponding to the method-body. The `returnType` is initialised to reflect the information available in the method signature, as resolved by the compiler. However, once the method-body is abstract-interpretted, the value of `returnType` is modified to reflect the reality. The return-type reflected in the value assigned to the `nodeName` continues to remain as resolved by the compiler. The `paramList`: its value is derieved from either the method-definition, or using the RTTI

types at the point of method invocation. Finally, the value of the `nodeName` attribute is computed from all the above. (Making an exception here might help: that the *declared* return-type of the method be used to construct the value of the `nodeName` rather than the RTTI return-type). This is the value used as the 'key' to store this node in the either of the two maps discussed below. The `fillParamList` method referenced below is a utility method that updates the instance attribute `paramList`, and returns its size.

The two system-assertions relevant here are: A. that the `nodeName` attribute of a `CodeBlob` object is mutually consistant with the values assigned to `containerType`, `noOfParams`, and `paramList`, and *vise versa*, and B. That `nodeName` is populated exactly once, during object creation. The value of its `returnType` attribute, however, is re-assigned upon abstract-interpretation of its behavioural-Spec. The instance-method below helps to preserve these system-assertions, by being *private*, and by *assert*-ing, at the outset, that the current-value of `nodeName` is Null, and, given that the other attributes are also *private*. `setNodeName` below checks for a '#' in the name to exit, if `this` node corresponds to a *synchronized*-block, rather than a method-body.

⟨ *CodeBlob Instance Methods* 27a ⟩ ≡

```
private void setNodeName (String nm) {
    assert (nodeName == null);
    nodeName = new MethodSignature (nm.intern());
    if (nm.indexOf('#') > 0)
        return;
}
```

◇

Fragment defined by 27a, 32b, 36b, 40, 41, 47, 50, 55.
Fragment referenced in 6.

⟨ *CodeBlob Instance Attributes* 27b ⟩ ≡

```
public Collection    rtReturnTypes      = null;
public TypeSignature getContainerType() { return nodeName.className(); };
public TypeSignature getCTReturnType()  { return nodeName.retType();   };
public String        getNodeName()      { return nodeName.signature(); };
public TypeSignature[] getParams()      { return nodeName.paramList(); };
public int           getParamCount()    { return nodeName.paramCount();};
```

◇

Fragment defined by 26, 27b, 28bc, 36a, 46b, 52b.
Fragment referenced in 6.

The analysis maintains two hash-Maps, `definedCode`, and `sCode`; both of which maintain a (1-1) map of method-signatures to corresponding `CodeBlob` nodes. `definedCode` stores all nodes that the fetched for analysis from either of source-1, or source-2, whereas, `sCode` stores only the subset corresponding to code that is marked *synchronized* in the input. The nodes that are on `sCode`, are also on `definedCode`.

⟨ *CodeBlob Class Attributes* 28a ⟩ ≡

```
protected static ConcurrentHashMap definedCode = new ConcurrentHashMap(); // defined
public    static ConcurrentHashMap sCode = new ConcurrentHashMap(); // Synchronized
```

◇

Fragment defined by 28a, 33a, 35a, 46a, 51, 98a.
Fragment referenced in 6.

The analysis creates a node for every method whose behaviour is either *defined* in the input, or can be *interpreted* from some such definition; Of these, the *defined* nodes are inserted into the `definedCode` map with its method-signature as the key, whereas the *interpretted* nodes are inserted into the hashMap that hangs off some node on the `definedCode`, (or `tobeDefinedCode`) map. Also methods that are *invoked* by *defined*-methods are stored in `definedCode`. Nodes on `definedCode` can, each, therefore, exist at three different levels of analysis: *Null*, *CompileTimeType*, and *RunTimeType*. Nodes corresponding to *invoked* methods whose definitions are not (yet) seen/interpreted are created and stored as place-holders: they are initialised to be at *Null* analysis. For any such node, (or for some fresh method), whose definition is seen in the input, the corresponding node is either upgraded (or initialized) to be at *CompileTimeType* analysis (and, of-course, the analysis of the corresponding method-definition seen in the input is conducted at that level). Unless, if the definition seen in the input corresponds to code marked as being *synchronized*: such nodes are upgraded or initialized with their analysis level set to *RunTimeType*, and the corresponding code is analysed accordingly.

⟨ *CodeBlob Instance Attributes* 28b ⟩ ≡

```
public enum AnalysisLevel {Null, CompileTimeType, RunTimeType };
public      AnalysisLevel accuracy = AnalysisLevel.Null;
```

◇

Fragment defined by 26, 27b, 28bc, 36a, 46b, 52b.
Fragment referenced in 6.

Since our OO-analysis un-folds the behavioural specification of a method-body to uncover its various interpretations based upon the actual types of the arguments supplied during its different invocations, and also based upon the actual type of the target-object used to invoke the instance mathod, we therefore maintain a Map of behavioural-varients of *defined* method-bodies, and also of *tobeDefined* method-signatres. So also, we store a reference back from such an interpretational-varient-node at analysis `accuracy` set to `Null`-level to its behavioural-spec node:

⟨ *CodeBlob Instance Attributes* 28c ⟩ ≡

```
public CodeBlob behaviourSpec = null;
```

◇

Fragment defined by 26, 27b, 28bc, 36a, 46b, 52b.
Fragment referenced in 6.

Step 1.b: Creating a node of the call-graph

Recollect from an earlier discussion that there are three sources from which method-signatures are fetched; In this subsection, below, we develop constructors to create nodes at each of these sources. They must preserve the program-invarients of the analysis, requiring that there to be at-most one `CodeBlob` instance corresponding to every method-definition (in the input-source), and also to its specific behavioral-varient (corresponding to some invocation of it, given its calling-context), that the compiler resolves against a given method-signature (its `behaviourSpec`). Methods whose behaviour is not specified in the input are discussed in the subsequent sub-section. Since the constructors, below, can-not return error, their correct invocation must therefore be conditional. When incorrectly invoked, however, they detect the inconsistancy being (attempted to be) created and assert failure.

The constructor below is intended for use with method-signatures fetched from source-2, when the invocation of a thus-far unknown method is first encountered. It accepts only a single argument, namely the (fully-qualified) name of the invoked method. Typical use of this constructor is to create the node corresponding to an *invoked* method, while the invoking method-body is being analysed at the `CompileTimeType`-level.

⟨ *CodeBlob Constructors* 29a ⟩ ≡
```
public CodeBlob (String mSign) {
    ⟨ Assert nodeName is not used-up (29b mSign ) 33b ⟩
    assert (mSign.indexOf ('#') < 0);
    setNodeName (mSign);
    System.err.println ("ZZZ: Created defined Node: " + mSign);
    if (! exhaustedReadingInputJars)
        definedCode.put (mSign, this);
};
```
◇
Fragment defined by 29a, 30ab, 32a, 37.
Fragment referenced in 6.

When analysing a method-body at the `RunTimeType`-level one encounters cases where the run-time types of the arguments passed to method-invocation are sub-classes of some of the corresponding formal-parameter-types in the parameter-list for the method-definition resolved by the compiler. And, moreover, through our abstract interpretation of the method-body, we are able to know the actual type of the target-object, as well as the actual types of all the arguments. The calling method-body parser will want to hand-over all this information to the CodeBlob constructor. The following constructor is intended for such use, when instantiating `CodeBlob`-instances for method-signatures fetched from source-3. There are two cases to handle here: one where

the compiler-resolved method signature as fetched from the significant-comment is an abstract/interface method, and another case, where it happens to identify a concrete method. In the first case, the said method-signature is firstly inserted into the `tobeDefinedCode`, and then the constructor below is invoked, if necessary. The constructor below asserts that there is no other node with the same name as the one attempted to be created, hanging off the implementors hashmap corresponding to the same method-signature (as fetched from the source-2).

⟨ *CodeBlob Constructors* 30a ⟩ ≡

```
/*public CodeBlob (String mSign, String cbName) {
    setNodeName (cbName);
    this.behaviourSpec = mSign;
    ConcurrentHashMap implementors = (ConcurrentHashMap) tobeDefinedCode.get(mSign);
    if (implementors == null)
        tobeDefinedCode.put (mSign, implementors = new ConcurrentHashMap ());
    implementors.put (getNodeName().intern(), this);
}; */
```

◇

Fragment defined by 29a, 30ab, 32a, 37.
Fragment referenced in 6.

In the second case (where the compiler-resolved method-signature is known to be a (potential) concrete method), the following constructor for the RTTI-based method-signature is provided.

⟨ *CodeBlob Constructors* 30b ⟩ ≡

```
public CodeBlob (CodeBlob cttb, String cbName) {
    String newName = cbName + "-:varientOf:-" + cttb.getNodeName();
    assert (cttb != null  &&  !cttb.getNodeName().equals (cbName));
    assert (varients.get(newName) == null);
    setNodeName (cbName);
    this.behaviourSpec = cttb;  //.nodeName.signature();
    varients.put (newName, this);
    System.out.println ("NewlyCreated: " + newName);
    assert (!cttb.getNodeName().equals (getNodeName()));
};
```

◇

Fragment defined by 29a, 30ab, 32a, 37.
Fragment referenced in 6.

More specifically, when parsing the byte-code of method-bodies, the line that contains a method-invocation also contains the (in the form of the significant comment, our source-2) fully-specified name of compiler-resolved method that could get invoked. Depending upon the RTTI type of the target-object the actually invoked method could

be a virtual method provided by one of the sub-classes. The following utility method generates the fully-specified method-varient's method-signature based upon the RTTI info of the target-object, and the types of arguments passed to the invocation.

⟨ *Algorithmic components to transform the graphs, and analyse* 31a ⟩ ≡

```
public static String getVarientCbName (String mSign,
    String oType, String[] args) {
    int   colon = mSign.indexOf   (':');    assert (colon >= 0);
    String fsmN = mSign.substring (0, colon);
    String  prt = mSign.substring (colon + 1);
    String   mN = fsmN .substring (fsmN.lastIndexOf('.') + 1);
    String retT = prt  .substring (prt.lastIndexOf (')') + 1);
    String retV = oType + "." + mN + ":(";
    for (int i = 0; i < args.length; i++)
        retV += args[i];
    return retV + ")" + retT;
};
```

◇

Fragment defined by 31a, 34ab, 46c, 48, 53b.
Fragment referenced in 6.

An instance-method, defined below, uses the parenthesised encoded parameter-list (contained in the fully-specified method-signatures) to discover the value for the `paramList` attribute of the containing-object.

⟨ *CodeBlob Instance unnecessary Methods* 31b ⟩ ≡

```
private int fillParamList (String pp) {//parenthesized parameters
    assert (pp.charAt (0) == '(');
    assert (pp.charAt (pp.length()-1) == ')');
    LinkedList l = new LinkedList();
    int rv = 0;
    for (int i = 1;  i < pp.length()-1;  rv++) {// post-incr i when consumed.
        l.add (rv, new String (pp.charAt(i)=='[' ? "[" : ""));
        if (pp.charAt(i) == '[')  i++;
        if ("BCFISZDJ".indexOf(pp.charAt(i)) >= 0)
            l.set (rv, ((String)l.get(rv)) + pp.charAt (i++));
        else if (pp.charAt(i) == 'L') {
            l.set (rv, ((String)l.get(rv)) +
                        pp.substring (i, pp.indexOf(';', i) + 1));
            i = pp.indexOf (';', i) + 1;
        } else {
            System.out.println ("Offending type-symbol: " + pp + ":"+i+ "?");
            assert (false);
        }
    }
    paramList = new String [rv];
```

```
        for (int i = 0;  i < rv;  i++)
            paramList[i] = (String)l.get(i);
        l.clear();
        return rv;
    };
    ◇
```

Fragment never referenced.

As class-definitions are parsed, when nodes corresponding to method-definitions (source-1 of method-signatures) are to be created in the call-graph, the constructor (defined below) that accepts information from the method-header as parameter-values is to be used.

⟨ *CodeBlob Constructors* 32a ⟩ ≡

```
    public CodeBlob (String ms, boolean isStat,
                                boolean isSynch, boolean isPub)  {
        this (ms);
        isPublic = isPub;
        isStatic = isStat;
        ⟨ Take note of synchronized use 35b ⟩
    }
    ◇
```

Fragment defined by 29a, 30ab, 32a, 37.
Fragment referenced in 6.

Yet another method below, `upgradeCB` is provided for the case that the method-signature whose definition is *now* encountered (fetched from source-1) was already fetched from source-2 (invoked in the already parsed code) in the past.

⟨ *CodeBlob Instance Methods* 32b ⟩ ≡

```
    public void upgradeCB (boolean isStat,
        boolean isSynch, boolean isPub)  {
        assert (definedCode.get (nodeName.signature()) != null);
        isPublic = isPub;    isStatic = isStat;
        ⟨ Take note of synchronized use 35b ⟩
    }
    ◇
```

Fragment defined by 27a, 32b, 36b, 40, 41, 47, 50, 55.
Fragment referenced in 6.

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

Step 1.c: Process `native`, `abstract` & `interface` methods

The input contains *native* methods whose header information is available but their implementation (behaviour specification) is empty; We can therefore not record information regarding any invocations that such methods might make. Similarly, the library contains definitions of *interfaces*, or *abstract Classes*, containing *abstract Method* members. Interface methods and *abstract methods* have no implementation. Although there is this similarity between these various types of methods, never-the-less, there are also an important difference: that wherever the compiler has resolved some method-call to either an `interface` or an `abstract` method, at run-time, some concrete method will be located that will *provide* the behavioural-varient to service the call. Whereas in the case of a `native` method-invocation, the behaviour of the code-fragment that *services* that call is well and truely beyond the scope of our investigation. Therefore, we collect interface-names seperately, and interface/abstract methods in the `tobeDefinedCode`, and only the `native` method-signatures are recorded in the `notDefinedCode`, defined below:

⟨ *CodeBlob Class Attributes* 33a ⟩ ≡

```
public static Set  notDefinedCode = new HashSet();
public static Map tobeDefinedCode = new HashMap();
public static Map      interfaces = new HashMap();
public static Map        varients = new HashMap();
◇
```

Fragment defined by 28a, 33a, 35a, 46a, 51, 98a.
Fragment referenced in 6.

The interesting aspect about abstract or native methods is that their *no-implementations-specified*-nature is *not* inferable at the point of their *invocation*. Interface-methods, however, are invoked by a different 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 definition that we shall encounter later 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 `definedCode`. *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: `notDefinedCode`, or `tobeDefinedCode`.

⟨ *Assert nodeName is not used-up* 33b ⟩ ≡

```
assert ((definedCode.get(@1) == null)  &&
        !notDefinedCode.contains(@1)  &&  (tobeDefinedCode.get(@1) == null));
◇
```

Fragment referenced in 29a.

Therefore, as our analysis proceeds, we will encounter occassions when we need to remove tuples from the `definedCode`, and move only the name of the method into one of the two structures: `notDefinedCode`, or `tobeDefinedCode`. 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.

⟨ *Algorithmic components to transform the graphs, and analyse* 34a ⟩ ≡

```
public static boolean isOkToCreate (String mSign) {
    return definedCode.get(mSign)==null   && !notDefinedCode.contains(mSign) &&
           tobeDefinedCode.get(mSign)==null;
};
```

◇

Fragment defined by 31a, 34ab, 46c, 48, 53b.
Fragment referenced in 6.

The instance-method `enlistNativeMethod`, below, removes methods from the `definedCode` list when they are detected as being either *native*, or *abstract*. The links from the node to be deleted to other nodes that are known to be invoking it are cleared.

⟨ *Algorithmic components to transform the graphs, and analyse* 34b ⟩ ≡

```
public static void nativeOrAbstract (String ms, boolean isNat) {
    if ((isNat && notDefinedCode.contains(ms))  ||
        tobeDefinedCode.get(ms) != null)
        return;
    if (isNat)     notDefinedCode.add (ms);
    else       {  tobeDefinedCode.put (ms, new ConcurrentHashMap());
        System.out.println ("Inserted into tobeDefinedCode: " + ms); }
    CodeBlob rmv = (CodeBlob) definedCode.remove (ms);
    if (rmv != null)  {
        for (Iterator i = rmv.callingCode.values().iterator();  i.hasNext();  )  {
             CodeBlob caller = (CodeBlob) i.next();
             if (i != null  &&  definedCode.get(caller.getNodeName()) != null)
                caller.calledCode.remove(ms);
        }
        rmv.callingCode.clear();
    }
};

public static boolean ensureField (String cn, String fn, boolean isStatic, String ct
    if (ctt.length() == 1  &&  TypeSignature.basicEncoded.contains (ctt))
        return false;
    assert ((ctt.startsWith("L") || ctt.startsWith("["))  &&  ctt.indexOf('.') < 0);
    ClassInfo objT = (ClassInfo) classes.get (cn);
    if (objT == null)
        classes.put (cn, objT = new ClassInfo(cn, null));
```

```
        String attT = (String) (isStatic  ?  objT.classAttributes.get(fn)
                                          :  objT.instanceAttributes.get(fn));
        if (attT == null)
            if (isStatic)   objT.classAttributes.put(fn, ctt);
            else            objT.instanceAttributes.put(fn, ctt);
        return true;
    };
    ◇
```

Fragment defined by 31a, 34ab, 46c, 48, 53b.
Fragment referenced in 6.

Step 1.d: 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. One of the `CodeBlob` constructors defined 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.

⟨ *CodeBlob Class Attributes* 35a ⟩ ≡

```
    public static int     synchronizedCBs = 0;  // count        synchronized statements
    public static int staticSynchronizeds = 0;  // count static-synchronized methods
    public static int      totalLocals = 0;  // Accumulate the locals count
    public static int        ssaLocals = 0;  // Count local-array "stores".
    public static int invocationsCount = 0;  // Count invocations.
    public static int         newCount = 0;  // Count the various types of newOp.
    public static int invGraphEdgeCount= 0;  // Count the edges in the Inv. Graph.
    ◇
```

Fragment defined by 28a, 33a, 35a, 46a, 51, 98a.
Fragment referenced in 6.

⟨ *Take note of synchronized use* 35b ⟩ ≡

```
    if (isSynch)  {
        setLockType (nodeName.className().name() + (isStatic ? ".class" : ""));
        if (isStatic)  staticSynchronizeds++;
    }
    ◇
```

Fragment referenced in 32ab.

Step 1.e: 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 it is not represented in the call-graph, since it does not contribute to cycles in the TLOG (as we shall see later). There are two other cases that need to be handled: one in which the called method happens to be some (potential) concrete method, and another where the called method is known to be either an interface method or some abstract method. We provide two different member-methods below for these two different cases: one that takes a `CodeBlob` object-reference of the called-method as a parameter, and another that accepts the method-signature of the called method as a `String` parameter. The reason why we seperate these two cases in this way (that is visible to the programmers of `JavaBlob` and `CsharpBlob`) is that we do-not want the *implementation* of `CodeBlob` to instantiate objects of the call-graph. Ideally, all instantiation of nodes of the call-graph ought to be done by implementors of sub-classes of the `CodeBlob` class that is an abstract class.

We use two maps: `calledCode`, and `callingCode` to represent the two directed edges between nodes of the call-graph. They are both initialised to seperate empty `HashMap`s, and are updated 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.

⟨ *CodeBlob Instance Attributes* 36a ⟩ ≡

```
protected HashMap   calledCode = new HashMap();
ConcurrentHashMap  callingCode = new ConcurrentHashMap();
```

◇

Fragment defined by 26, 27b, 28bc, 36a, 46b, 52b.
Fragment referenced in 6.

⟨ *CodeBlob Instance Methods* 36b ⟩ ≡

```
public void noteCallTo (CodeBlob called)  {
    if (calledCode.keySet().contains(called.getNodeName()))
        return;
    called.callingCode.put (getNodeName(), this);
    calledCode.put (called.getNodeName(), called); invGraphEdgeCount++;
}
public void noteCallTo (String called)  {
    if (calledCode.keySet().contains(called) || notDefinedCode.contains(called))
        return;
    if (tobeDefinedCode.get (called) == null)
        tobeDefinedCode.put (called, new ConcurrentHashMap());
    calledCode.put (called, null);
}
public void noteCallTo (String crMsign, CodeBlob rtcb)  {
```

```
        ConcurrentHashMap hmp = null;
        if (notDefinedCode.contains(crMsign))
            return;
        if ((hmp = (ConcurrentHashMap) tobeDefinedCode.get (crMsign)) == null)
            tobeDefinedCode.put (crMsign, hmp = new ConcurrentHashMap ());
        calledCode.put (crMsign, rtcb);
        rtcb.callingCode.put (getNodeName(), this); invGraphEdgeCount++;
        hmp.put (rtcb.getNodeName(), rtcb);
    }
    public void noteCallTo (CodeBlob crcb, CodeBlob rtcb)  {
        calledCode.put (crcb.getNodeName(), rtcb);
        rtcb.callingCode.put (getNodeName(), this); invGraphEdgeCount++;
    }
    ◇
```

Fragment defined by 27a, 32b, 36b, 40, 41, 47, 50, 55.
Fragment referenced in 6.

Step 1.f: 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.g below. `sCode` stores 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 the code enclosed by 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 Constructors* 37 ⟩ ≡

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

```
        p.noteCallTo (this);
        behaviourSpec = p.behaviourSpec == null ? p : p.behaviourSpec;
        accuracy = AnalysisLevel.RunTimeType;
        synchronizedCBs++;
    }
    ◇
```

Fragment defined by 29a, 30ab, 32a, 37.
Fragment referenced in 6.

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

For methods that have `synchronized` blocks of code, it is necessary to conduct 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. After the end 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.h: Create *otLock* nodes of the TLOG

For every non-null value (say *lockName*) assigned to the `lockType` attribute of any `CodeBlob` object, we create an `OTlock` (standing for object/type lock) object and insert it into a global HashMap called `tlog`. Recall that *lockName* would be either like ”typeName”, or like ”typeName.Class”. Every object of the TLOG 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 coresponding type.

⟨ *The Type lock order graph* 38a ⟩ ≡

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

Fragment referenced in 8a.

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

⟨ *OTlock Instance Attributes* 38b ⟩ ≡

```
    String     lockName = null;
    String     version     = "1.7";
    HashMap    tlogEdges   = new HashMap();
    HashSet    lContenders = new HashSet();
    public OTlock (String name) {
        lockName = name; // Surprise: this attribute is never used.
        if (!name.endsWith (".class"))
            name = name + ".class";
        version  = getVersion (name);
    };
    private String getVersion (String name) {
        String retval = "1.7";
        try {
            Process c = Runtime.getRuntime().exec ("./getVersion " + name);
            BufferedReader in = new BufferedReader (
                                new InputStreamReader (c.getInputStream()));
            retval = in.readLine();
            if (c.waitFor() != 0)
                System.out.println ( "getVerion Failed: " + c.exitValue());
        } catch (Exception e) {
            System.out.println ("Exceotion in getVersion: " + e);
        }
        return retval;
    };
    ◇
```

Fragment referenced in 8a.

```
"getVersion" 39≡
      if grep --silent $1 classList/1.1/classes; then echo 1.1;
    elif grep --silent $1 classList/1.2/classes; then echo 1.2;
    elif grep --silent $1 classList/1.3/classes; then echo 1.3;
    elif grep --silent $1 classList/1.4/classes; then echo 1.4;
    elif grep --silent $1 classList/1.5/classes; then echo 1.5;
    elif grep --silent $1 classList/1.6/classes; then echo 1.6;
    fi
    ◇
```

The `CodeBlob` instance method `setLockType`, defined below, is used to club together the code that assigns a value to the `lockType` attribute of the `CodeBlob` object, and that which creates (where necessary) an object of type `OTlock` to represent the lock that the said code attempts to acquire before proceeding to compute. This public instance method to set the value for a private attribute, asserts that the `lockType` attribute value is null, and that the argument-value is non-null. This assertion enforces the programming discipline that the value of the private attribute can be set at-most

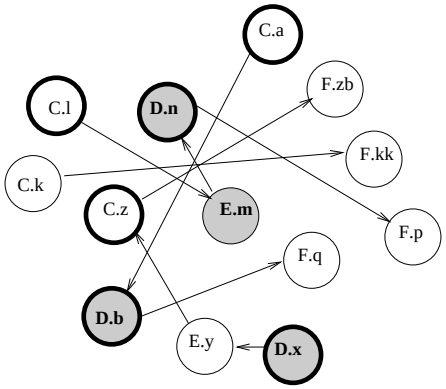

Figure 1.2: The graph constructed after Step 1.

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.

⟨ *CodeBlob Instance Methods* 40 ⟩ ≡

```
public void setLockType (String lock)  {
    assert (lockType == null  &&  lock != null);
    assert (!lock.equals ("com")  &&  !lock.equals("java")  &&
            !lock.equals ("sun")  &&  !lock.equals("javax"));
    lockType = lock;
    OTlock rv = (OTlock)OTlock.locks.get(lock);
    if (rv == null)
        OTlock.locks.put(lock, rv = new OTlock(lock));
    rv.lContenders.add (this);
    sCode.put (getNodeName(), this);
}

public String getLockType()     {  return lockType;  };

public abstract void processMethodBody() throws java.lang.Exception;
```

◇

Fragment defined by 27a, 32b, 36b, 40, 41, 47, 50, 55.
Fragment referenced in 6.

Step 2: Cleanse `definedCode` of undefined yet invoked methods

In the Step 1, above, we recorded invoked class methods into `definedCode` hoping to find their definitions, possibly later in the input-scan. After completing the input-scan in that step, we now need to cleanse the `definedCode` of invoked-method-nodes that are likely behavioural varients of inherited-code from parent-classes. Such nodes

that we remove from `definedCode`, we shall insert into the list of interpretational-`varients` of the inherited-method whose bahaviour is to be used as the specification when populating the varient-behaviour.

The way to tell that a node in the `definedCode` list is a candidate for such transfer, is to confirm that its `accuracy` attribute is set to `AnalysisLevel.Null`, and be able to find one of its parent-classes that does actually define a method with a compatible signature.

⟨ *CodeBlob Instance Methods* 41 ⟩ ≡

```
/****
public boolean inheritFromDefinedCode ()
    throws java.lang.Exception {
    if (accuracy == AnalysisLevel.Null)  {
        String clsNm = nodeName.className().name();
        String rest  = nodeName.dropClassNmGetRest();
        CodeBlob pc = null, ac = null;
        for (ClassInfo ci = null;  clsNm != null;
             clsNm = ((ci = (ClassInfo)classes.get(clsNm)) == null) ?
                     null : ci.parentType) {
             String bspecNm = clsNm +"."+ rest;
             if (notDefinedCode.contains (bspecNm)) {
                 assert (    definedCode.get(bspecNm) == null  &&
                          tobeDefinedCode.get(bspecNm) == null);
                 notDefinedCode.add (c.getNodeName());
                 definedCode.remove (c.getNodeName());
                 for (Iterator j = callingCode.values().iterator();  j.hasNext();  )
                   CodeBlob caller = (CodeBlob) j.next();
                   if (caller != null  &&  definedCode.get(caller.getNodeName()) != n
                     caller.calledCode.remove(getNodeName());
                 }
                 callingCode.clear();
                 break;
             } else if (tobeDefinedCode.get(bspecNm) != null) {
                 assert (   definedCode.get(bspecNm) == null  &&
                         !notDefinedCode.contains(bspecNm));
                 tobeDefinedCode.put (getNodeName(), new ConcurrentHashMap());
                 definedCode.remove (c.getNodeName());
                 for (Iterator j = callingCode.values().iterator();  j.hasNext();  )
                   CodeBlob caller = (CodeBlob) j.next();
                   if (caller != null  &&
                     definedCode.get(caller.getNodeName()) != null) {
                     caller.calledCode.remove(getNodeName());
                     caller.calledCode.put (getNodeName(), null);
                   }
                 }
                 callingCode.clear();
                 break;
             } else if ((pc = (CodeBlob) definedCode.get(bspecNm)) != null)
                 if (pc.accuracy != AnalysisLevel.Null) {
```

```
                    c.upgradeToCTTAnalysis (pc);
                break;
             }
          }
      }
      if (definedCode.get (c.getNodeName()) == c  &&
        c.accuracy == AnalysisLevel.Null)  {
        String clsNm = c.nodeName.className().name();
        String rest  = c.nodeName.dropClassNmGetRest();
        ClassInfo ci = (ClassInfo) classes.get (clsNm);
        List ii = ci != null ? ci.getInterfacesImplemented()
                             : getInterfacesExtendedBy (clsNm);
        // System.out.println ("Looking for undefined: " + c.getNodeName());
        if (ii != null  &&  ii.size() > 0) {
          for (int k = 0;  k < ii.size();  k++) {
            String ipNm = null;
            ipNm = (String) ii.get(k) + "." + rest;
            // System.out.println ("Checking: " + ipNm + " against interface: " + ii.g
            if (tobeDefinedCode.get (ipNm) != null) {
              assert (     definedCode.get(ipNm) == null  &&
                     ! notDefinedCode.contains(ipNm));
              tobeDefinedCode.put (c.getNodeName(), new ConcurrentHashMap());
              assert (c == (CodeBlob) definedCode.remove (c.getNodeName()));
              for (Iterator j = c.callingCode.values().iterator();  j.hasNext();  )  {
                CodeBlob caller = (CodeBlob) j.next();
                if (caller != null  &&
                  definedCode.get(caller.getNodeName()) != null) {
                  caller.calledCode.remove(c.getNodeName());
                  caller.calledCode.put (c.getNodeName(), null);
                }
              }
              c.callingCode.clear();
              // System.out.println ("Able to find undefined: " + ipNm);
              break;
            }
          }
        }
      }
      if (definedCode.get (c.getNodeName()) == c  &&
        c.accuracy == AnalysisLevel.Null)
        System.out.println ("XXX Undefined: " + c.getNodeName());
  };
  ****/
  public static void inheritFromDefinedCode ()
      throws java.lang.Exception {
      for (Iterator i = definedCode.values().iterator();  i.hasNext();  )  {
          CodeBlob c = (CodeBlob) i.next();
          if (c != null  &&  c.accuracy == AnalysisLevel.Null)  {
              String clsNm = c.nodeName.className().name();
              String rest  = c.nodeName.dropClassNmGetRest();
```

```
    CodeBlob pc = null, ac = null;
    for (ClassInfo ci = null;  clsNm != null;
         clsNm = ((ci = (ClassInfo)classes.get(clsNm)) == null) ?
                 null : ci.parentType) {
         String bspecNm = clsNm +"."+ rest;
         if (notDefinedCode.contains (bspecNm)) {
             assert (    definedCode.get(bspecNm) == null  &&
                     tobeDefinedCode.get(bspecNm) == null);
             notDefinedCode.add (c.getNodeName());
             assert (c == (CodeBlob) definedCode.remove (c.getNodeName()));
             for (Iterator j = c.callingCode.values().iterator();  j.hasNext
               CodeBlob caller = (CodeBlob) j.next();
               if (caller != null  &&  definedCode.get(caller.getNodeName())
                 caller.calledCode.remove(c.getNodeName());
             }
             c.callingCode.clear();
             break;
         } else if (tobeDefinedCode.get (bspecNm) != null) {
             assert (     definedCode.get(bspecNm) == null  &&
                     ! notDefinedCode.contains(bspecNm));
             tobeDefinedCode.put (c.getNodeName(), new ConcurrentHashMap());
             assert (c == (CodeBlob) definedCode.remove (c.getNodeName()));
             for (Iterator j = c.callingCode.values().iterator();  j.hasNext
               CodeBlob caller = (CodeBlob) j.next();
               if (caller != null  &&
                 definedCode.get(caller.getNodeName()) != null) {
                 caller.calledCode.remove(c.getNodeName());
                 caller.calledCode.put (c.getNodeName(), null);
               }
             }
             c.callingCode.clear();
             break;
         } else if ((pc = (CodeBlob) definedCode.get(bspecNm)) != null)
             if (pc.accuracy != AnalysisLevel.Null) {
                 c.upgradeToCTTAnalysis (pc);
             break;
         }
    }
}
if (definedCode.get (c.getNodeName()) == c  &&
  c.accuracy == AnalysisLevel.Null)  {
  String clsNm = c.nodeName.className().name();
  String rest  = c.nodeName.dropClassNmGetRest();
  ClassInfo ci = (ClassInfo) classes.get (clsNm);
  List ii = ci != null ? ci.getInterfacesImplemented()
                       : getInterfacesExtendedBy (clsNm);
  // System.out.println ("Looking for undefined: " + c.getNodeName());
  if (ii != null  &&  ii.size() > 0) {
    for (int k = 0;  k < ii.size();  k++) {
      String ipNm = null;
```

```
                ipNm = (String) ii.get(k) + "." + rest;
                // System.out.println ("Checking: " + ipNm + " against interface: " + i
                if (tobeDefinedCode.get (ipNm) != null) {
                  assert (     definedCode.get(ipNm) == null  &&
                          ! notDefinedCode.contains(ipNm));
                  tobeDefinedCode.put (c.getNodeName(), new ConcurrentHashMap());
                  assert (c == (CodeBlob) definedCode.remove (c.getNodeName()));
                  for (Iterator j = c.callingCode.values().iterator();  j.hasNext();
                    CodeBlob caller = (CodeBlob) j.next();
                    if (caller != null  &&
                      definedCode.get(caller.getNodeName()) != null) {
                      caller.calledCode.remove(c.getNodeName());
                      caller.calledCode.put (c.getNodeName(), null);
                    }
                  }
                  c.callingCode.clear();
                  // System.out.println ("Able to find undefined: " + ipNm);
                  break;
                }
              }
            }
          }
          if (definedCode.get (c.getNodeName()) == c  &&
            c.accuracy == AnalysisLevel.Null)
            System.out.println ("XXX Undefined: " + c.getNodeName());
          else  if (definedCode.get (c.getNodeName()) == null)
            System.out.println ("XXX Removed from Defined Code: " + c.getNodeName());
      }
  };

  public static CodeBlob getCodeOrVarientBlob (String rcms, String rms) {
      CodeBlob rv = null;
      if (rcms.indexOf(".\"<init>\":") > 0  ||  rcms.equals(rms))
          rv = (CodeBlob) definedCode.get (rcms);
      else
          rv = (CodeBlob) varients.get (rms +"-:varientOf:-"+ rcms);
      if (rv == null) System.err.println ("ZZZ: Searched for: " + rcms + ":" +
                                          rms + "\n\tBut returned Null");
      return rv;
  };
  //public static void cleanseDefinedCode ()
  //    throws java.lang.Exception {
  //    for (Iterator i = definedCode.values().iterator();  i.hasNext();  )  {
  //        CodeBlob c = (CodeBlob) i.next();
  //        if (c != null  &&  c.accuracy == AnalysisLevel.Null)  {
  //            i.remove ();
  //            if (noteVarientAndGetBehaviourSpec (c) == null)
  //                System.out.println ("XXX Deleted undefined-Node: " +
  //                                    c.getNodeName());
  //        }
```

```
//    }
//};
//            if (arg instanceof CodeBlob)
//                if ((ac = (CodeBlob) pc.varients.get (cnns)) != null) {
//                    ((CodeBlob)arg).setRightCallerNdCalledNodes (cnns, ac);
//                    ac.calledCode .putAll (((CodeBlob)arg).calledCode);
//                    ac.callingCode.putAll (((CodeBlob)arg).callingCode);
//                    assert (ac.behaviourSpec == pc); //.nodeName.signature());
//                } else {
//                    ((CodeBlob)arg).behaviourSpec = pc; // .nodeName.signature();
//                    pc.varients.put (cnns, arg);
//                }

public static CodeBlob getBspecForIVcall (MethodSignature ms)  { // Interface or Vir
    // null-return implies that the behaviour provider is a native method ...
    String cN = ms.className().name();
    if (interfaces.keySet().contains(cN)  ||
        tobeDefinedCode.get(ms.signature()) != null)
        return null; // ... or an interface method -- not to some implementing class
    String rest = ms.dropClassNmGetRest();
    CodeBlob pc = null, ac = null;
    for (ClassInfo ci = null;  cN != null;
         cN = ((ci = (ClassInfo)classes.get(cN)) == null) ? null : ci.parentType) {
        String lookinFor = cN +"."+ rest;
        if ((pc = (CodeBlob) definedCode.get (lookinFor)) != null)
            return pc;
        else if (notDefinedCode.contains (lookinFor))
            return null;  // Behaviour provider is a native method !!
        else if (tobeDefinedCode.get (lookinFor) != null) {
            System.out.println ("Binding against Abstract Method: " + lookinFor);
            return null;
        }
    }
    System.out.println ("Behaviour Spec missing: " + ms.signature());
    return null;     // this makes the compiler not-to-complain.
}
◇
```

Fragment defined by 27a, 32b, 36b, 40, 41, 47, 50, 55.
Fragment referenced in 6.

Step 3: Compute the `startPoints` set

In this step, we iterate over the members of `sCode`, collecting all their *sufficiently close, concrete* `public` invokers into their (respective) `concPublicCallers` instance-attribute. We define a magic-number called `fanout` that decides the amount of *distance*, how-much further *out* we go looking for such a *concrete* `public` invoker of the `synchronized` method. While we collect the above information, we also *upgrade*

the analysis-level of the nodes that contribute to these sets (or are encountered along the call-path) to `RunTimeType`-Level. Below, we arbitrarily choose some value for `fanout`, to limit the fan-out, and also to avert any cycles in the call-graph from causing infinite-loops. Here we are interested in listing only such invokers that have not already locked some other type/object on the way to invoking this current synchronized method, therefore, that we do not chase invokers that are themselves `synchronized`.

⟨ *CodeBlob Class Attributes* 46a ⟩ ≡
```
static final int fanout = 5;
```
◇

Fragment defined by 28a, 33a, 35a, 46a, 51, 98a.
Fragment referenced in 6.

⟨ *CodeBlob Instance Attributes* 46b ⟩ ≡
```
Set  concPublicInvokers = null;
```
◇

Fragment defined by 26, 27b, 28bc, 36a, 46b, 52b.
Fragment referenced in 6.

⟨ *Algorithmic components to transform the graphs, and analyse* 46c ⟩ ≡
```
public static void collectStartPoints () // BUP
    throws java.lang.Exception {
    for (Iterator i = sCode.values().iterator();  i.hasNext();  )  {
        CodeBlob n = (CodeBlob) i.next();
        // if (!n.isPublic  ||  !n.isConcrete())
            n.concPublicInvokers = n.collectPublicInvokers(fanout);
    }
};
public static List getInterfacesExtendedBy (String iNm) {
    List rv = new java.util.ArrayList ();
    rv.add (iNm);
    for (int i = 0;  i < rv.size();  i++)  {
        String   ii = (String) rv.get(i);
        String[] ie = null;
        if ((ie = (String[]) CodeBlob.interfaces.get(ii)) != null)
            for (int j = 0;  j < ie.length;  j++)
                rv.add (ie[j]);
    }
    return rv;
};

public static void countStartPoints ()  throws java.lang.Exception {
    int retval = 0;
    for (Iterator i = definedCode.values().iterator();  i.hasNext();  )  {
        CodeBlob n = (CodeBlob) i.next();
```

```
            if (n != null  &&  (n.isPublic  &&  n.isConcrete()))
                retval ++;
        }
        System.out.println ("Public Concrete StartPoints = " + retval);
    };
    public static void reportSubclassingNdImplements ()  throws java.lang.Exception {
        HashSet cS = null, iS = null;
        String  cN = null, iN = null;
        int subClassingIncidence = 0, interfaceImplementationIncidence = 0;
        for (Iterator i = classes.keySet().iterator();  i.hasNext();  )  {
            cN = (String) i.next();
            cS = new HashSet(); cS.add (cN);
            iS = new HashSet();
            for (ClassInfo ci = null;  cN != null;
                 cN = ((ci = (ClassInfo)classes.get(cN)) == null) ?
                  null : ci.parentType)  {
                if (cN != null) cS.add (cN);
                if (ci != null) iS.addAll (ci.getInterfacesImplemented ());
            }
            for (Iterator k = iS.iterator();  k.hasNext();  ) {
                iN = (String) k.next();
                iS.addAll (getInterfacesExtendedBy (iN));
            }
            subClassingIncidence += (cS.size()-1);
            interfaceImplementationIncidence += iS.size();
        }
        System.out.println ("Incidence of Subclassing: " + subClassingIncidence);
        System.out.println ("Incidence of Interface Implementation: " + interfaceImpleme
    };

    //public static void collectStartPointsTDN ()  throws java.lang.Exception {
    //    for (Iterator i = definedCode.values().iterator();  i.hasNext();  )  {
    //        CodeBlob n = (CodeBlob) i.next();
    //        if (n != null  &&  (n.isPublic  ||  n.isConcrete()))
    //            retval ++;
    //        if (!n.isPublic  ||  !n.isConcrete())
    //            n.concPublicInvokers = n.collectPublicInvokers(fanout);
    //    }
    //};
    ◇
```

Fragment defined by 31a, 34ab, 46c, 48, 53b.
Fragment referenced in 6.

⟨ *CodeBlob Instance Methods* 47 ⟩ ≡

```
    public Set collectPublicInvokers (int countDown)
        throws java.lang.Exception {
        Set rv = new HashSet();
        rv.add (this);
```

```
        if (countDown == 0  ||  callingCode == null)
            return rv;
        for (Iterator i = callingCode.values().iterator();  i.hasNext();  )  {
            CodeBlob c = (CodeBlob) i.next();
            if (c != null)  {
                if (c.isPublic  &&  c.isConcrete())
                    rv.add (c);
                else if (c.getLockType() == null)  {
                    Set temp = c.collectPublicInvokers(countDown - 1);
                    if (temp != null  &&  temp.size() > 0)
                        rv.addAll (temp);
                }
            }
        }
        return rv;
    }
    ◇
```

Fragment defined by 27a, 32b, 36b, 40, 41, 47, 50, 55.
Fragment referenced in 6.

Step 4: Attempt to increase cardinality of the TLOG

Analyse all *varient-behaviour* nodes hanging off all the `synchronized` nodes in the `sCode` map, at the *RunTimeType*-level. Consider, for instance, that an (inherited) `synchronized` method *sa()* is invoked upon an object of type *B*, which inherits it from its parent-class (say) *A*. The method signature *B::sa()V*, therefore, is generated by the `RunTimeType`-Level analysis, to denote the *behavioural-varient* of concrete method *A::sa()V*. The TLOG that we would have been populating while parsing the input source contains node A, but not node B, yet. The purpose of this step is to analyse (at the `RunTimeType`-Level) the method-body of *A::sa()V* to generate all its behavioural-varients like *B::sa()V* so as to cause additional nodes, like node B, to get populated into the TLOG.

While executing this step, it is possible that more nodes get created, corresponding to behavioural-varients of `synchronized` code. Such nodes, ideally, should also get `RunTimeType`-analysed, for best affect. Care should be exercised, not to miss-out on any of such, if this is attempted. The appropriate naming scheme, in such a case, is to use the `nodeName` of the behavioural-spec node as the prefix, some appropriate seperator-symbol, followed by the `nodeName` of the behavioural-varient. The first method below, delegates the task of actual `RealTimeType`-analysis (through abstract-interpretation) to another method: `rttAnalyseVarient`.

⟨ *Algorithmic components to transform the graphs, and analyse* 48 ⟩ ≡

```
    //static public void interpretSynchNdVarients ()
    //    throws java.lang.Exception {
    //    for (Iterator i = sCode.entrySet().iterator();  i.hasNext();  )  {
    //        Map.Entry sMe = (Map.Entry)  i.next();
```

```
//        CodeBlob  sM  = (CodeBlob) sMe.getValue();
//        assert (sM != null  &&  sM.getLockType() != null);
//        if (sM.nodeName.signature().indexOf('#') > 0)
//            continue;
//        if (sM.accuracy != AnalysisLevel.RunTimeType) {
//            CodeBlob mSm = sM.upgradeToRTTAnalysis ();
//            if (mSm != null  &&  mSm != sM)
//                sMe.setValue (mSm);
//        }
//        if (sM.varients == null  ||  sM.varients.size() == 0)
//            continue;
//        for (Iterator j = sM.varients.entrySet().iterator();  j.hasNext();  )  {
//            Map.Entry sVp = (Map.Entry) j.next();
//            CodeBlob sV = (CodeBlob) sVp.getValue();
//            assert (sV.behaviourSpec != null);
//            System.out.println ("Requesting Upgrade:\nBase NodeName: " +
//                sM.getNodeName() + "\nVarient  Name: " + sV.getNodeName());
//            CodeBlob cV = sV.upgradeToRTTAnalysis ();
//            if (cV != sV  &&  cV != null)
//                sVp.setValue (cV);
//        }
//    }
//}

//static public void interpretInterfaceVarients ()  throws java.lang.Exception {
//    for (Iterator i = tobeDefinedCode.entrySet().iterator();  i.hasNext();  )  {
//        Map.Entry        iiM  = (Map.Entry)         i.next();
//        ConcurrentHashMap iM  = (ConcurrentHashMap) iiM.getValue();
//        assert (iM != null);
//        if (iM.size() == 0)    continue;
//        System.out.println ("Interface: " + ((String)iiM.getKey()));
//        for (Iterator j = iM.entrySet().iterator();  j.hasNext();  )  {
//            Map.Entry impe = (Map.Entry) j.next();
//            CodeBlob imp = (CodeBlob) impe.getValue();
//            assert (imp != null  &&  imp.behaviourSpec != null);
//            System.out.println ("Requesting Interpretation:\nEntry-Key: " +
//                impe.getKey() + "\nRT-Method: " + imp.getNodeName());
//            CodeBlob cV = imp.upgradeToRTTAnalysis ();
//            if (cV != imp  &&  cV != null)
//                System.out.println ("Creating disconnect");
//                // XXX: this is not completely done, yet??
//        }
//
//    }
//}
◇
```

Fragment defined by 31a, 34ab, 46c, 48, 53b.
Fragment referenced in 6.

⟨ *CodeBlob Instance Methods* 50 ⟩ ≡

```
    public CodeBlob upgradeToRTTAnalysis ()
        throws java.lang.Exception {
        if (behaviourSpec == null  &&  definedCode.get(getNodeName()) == this) {
            upgradeToRTTAnalysis (this);
            return null;
        }
        if (behaviourSpec == null)
            System.out.println ("Howcome behaviourSpec is null: " + getNodeName());
        assert ((behaviourSpec == null)  ||  (behaviourSpec instanceof CodeBlob));
        //assert (behaviourSpec != null  &&
        //        behaviourSpec.accuracy != AnalysisLevel.Null);
        // if (pcb.accuracy != AnalysisLevel.Null)
            upgradeToRTTAnalysis (behaviourSpec);
        return this;
        /* String bspec = //(behaviourSpec instanceof CodeBlob  ?
                   ((CodeBlob)behaviourSpec).getNodeName();// : (String) behaviourSpec);
        String rest  = bspec.substring (bspec.lastIndexOf("."));
        String clsNm = getContainerType().name();
        for (ClassInfo ci = null;  clsNm != null;
             clsNm = ((ci = (ClassInfo)classes.get(clsNm)) == null) ?
              null : ci.parentType) {
            CodeBlob pcb = (CodeBlob) definedCode.get (clsNm + rest);
            if (pcb != null) {
                if (pcb.getNodeName().equals (getNodeName())) {
                    setRightCallerNdCalledNodes (getNodeName(), pcb);
                    pcb.calledCode .putAll (calledCode);
                    pcb.callingCode.putAll (callingCode);
                    if (pcb.accuracy != AnalysisLevel.RunTimeType)
                        pcb.upgradeToRTTAnalysis (pcb);
                    return pcb;
                } else {
                }
            }
        }
        return null;*/
    };

    public void setRightCallerNdCalledNodes (String name, CodeBlob node) {
        for (Iterator i = calledCode.entrySet().iterator();  i.hasNext();  ) {
            Map.Entry j = (Map.Entry) i.next();
            if (((CodeBlob)j.getValue()).getNodeName().equals (name))
                j.setValue (node);
        }
        for (Iterator i = callingCode.entrySet().iterator();  i.hasNext();  ) {
            Map.Entry j = (Map.Entry) i.next();
            if (((CodeBlob)j.getValue()).getNodeName().equals (name))
                j.setValue (node);
        }
    };
```

```
public static int rttAnalysisCount = 0, cttAnalysisCount = 0;
abstract public void upgradeToRTTAnalysis (CodeBlob spec)
    throws java.lang.Exception;
abstract public void upgradeToCTTAnalysis (CodeBlob spec)
    throws java.lang.Exception;
```
◇

Fragment defined by 27a, 32b, 36b, 40, 41, 47, 50, 55.
Fragment referenced in 6.

The `defineVarient` is an instance method, since it (effectively) populates the `CodeBlob` instance corresponding to the hetherto undefined method, using the argument-method-body as a specification.

⟨ *CodeBlob Class Attributes* 51 ⟩ ≡
```
static protected HashMap classes = new HashMap();
/* This class-attribute stores a ClassInfo object corresponding
 *  to every class encountered while parsing the input, mapping
 *  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 parent-class-name. The Java \& C-Sharp
 *  applications that inherit from \verb|CodeBlob| must populate
 *  this data, using calls like:
 *  classes.put (subClass, ClassInfo.getObject (subClass, parentType));
 */

public boolean isConcrete () {
    ClassInfo cinfo = ((ClassInfo) classes.get(getContainerType().name()));
    if (cinfo != null  &&  cinfo.isAbstract)
        return false;
    if (interfaces.keySet().contains (getContainerType().name()))
        return false;
    for (int i = 0;  i < getParamCount();  i++)  {
        String nm = getParams()[i].name();
        ClassInfo c = (ClassInfo) classes.get (nm);
        if (interfaces.keySet().contains(nm)  ||  c == null  ||  c.isAbstract)
            return false;
    }
    return true;
};
```
◇

Fragment defined by 28a, 33a, 35a, 46a, 51, 98a.
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.

⟨ *ClassInfo Instance Attributes* 52a ⟩ ≡
```
public String  typeName   = null;
public String  parentType = null;
public boolean isAbstract = false;
```
◇

Fragment defined by 52a, 61a.
Fragment referenced in 8b.

Step 5: Discover *reachability* within the `sCode` set

Iterate over nodes of the `sCode` set, including all their behavioural-varients, populating their `sEdges` attribute (type: `Set`) as follows: start simulating the call-stack from (for each of them) their public-concrete-invokers, inserting a directed-edge between two nodes of this (`sCode`) 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 no other non-`sCode` code-blobs in the call-stack, in between the stack-frames corresponding to the caller-method, & the called-method. Drop any path from further investigation if the first lock aquired along it happens *not* to be the one indicated by the `lockType` of the specific `sCode`-member being investigated.

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 `synchronized` nodes, we instantiate a set to hold the result of the computation. Accordingly, a class-static method, and an instance method are specified below that compute the value for this attribute. The 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 4. The bold edges correspond to the newly-added `sEdges`.

⟨ *CodeBlob Instance Attributes* 52b ⟩ ≡
```
HashSet sEdges = null;
```
◇

Fragment defined by 26, 27b, 28bc, 36a, 46b, 52b.
Fragment referenced in 6.

The Class-static method, below, starts with reporting some statics regarding the input to our analysis, followed by a simple iteration over members of the `sCode` set, invoking the instance method for each, that populates the `sEdges` attribute. Notice the use of a `workArea` that stores the set of nodes visited along a path, so as to avoid ending up in some infinite-loop along some cycle in the underlying call-graph. invocationsCount

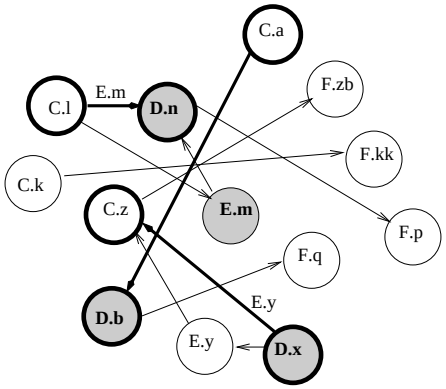

Figure 1.3: The graph after executing Step 3.

⟨ *Report Statistics* 53a ⟩ ≡

```
System.err.println (
        "\n        Method invocation count:\t" +        invocationsCount +
        "\n                    New count:\t" +               newCount +
        "\n            Total locals count:\t" +            totalLocals +
        "\n              SSA locals count:\t" +              ssaLocals +
        "\nDefined or invoked method count:\t" +     definedCode.size() +
        "\n    Invocation graph Edge count:\t" +       invocationsCount +
        "\n Defined          Varient count:\t" +        varients.size() +
        "\n           Native method count:\t" +   notDefinedCode.size() +
        "\n  Abstract or I/f  method count:\t" +  tobeDefinedCode.size() +
        "\n  Abstract or I/f Varient count:\t" + abstrctIntrfceVarient() +
        "\n                    Class count:\t" + classes.keySet().size() +
        "\n               Interfaces count:\t" +       interfaces.size() +
        "\n Synch code-blob & method count:\t" +           sCode.size() +
        "\n Synch static      method count:\t" +    staticSynchronizeds +
        "\n Synch          Code-Blob count:\t" +        synchronizedCBs +
        "\n Synch Code       Varient count:\t" +    synchVarientCount() +
        "\n Synch Code         Total count:\t" +           sCode.size() +
        "\n               size of the tlog:\t" +   OTlock.locks.keySet().size() +
        "\n Run-Time   Type-Analysed count:\t" +       rttAnalysisCount);
```

◇

Fragment referenced in 70.

⟨ *Algorithmic components to transform the graphs, and analyse* 53b ⟩ ≡

```
public static void sCodeReachedFromSCode ()
    throws java.lang.Exception {
    int rv = 0, rv1 = 0, rv2 = 0;
    HashSet avoidRepeats = new HashSet();
    for (Iterator i = sCode.values().iterator();  i.hasNext();  )  {
```

```
        CodeBlob mB = (CodeBlob) i.next();
        assert (mB != null);
        mB.concPublicInvokers = mB.collectPublicInvokers(fanout);
        if (mB.concPublicInvokers != null) {
            rv++;
            // rv1 += mB.concPublicInvokers.size();
            for (Iterator j = mB.concPublicInvokers.iterator();  j.hasNext();  )  {
                CodeBlob spcB = (CodeBlob) j.next();
                if (! avoidRepeats.add (spcB.getReportableName(null)))
                    continue;
                rv1 ++;
                HashSet workArea = new HashSet();
                workArea.add (spcB.getNodeName());
                spcB.augmentReachableSCode (mB.sEdges = new HashSet(),
                       workArea, mB.getLockType(), spcB.getReportableName(null));
                rv2 += mB.sEdges.size();
            }
        }
    }
    System.out.println ("Total number of Synchronized methods investigated = " + rv)
    System.out.println ("Total Concurrent public invokers investigated = " + rv1);
    System.out.println ("Total sEdges discovered  = " + rv2);
}
public static void computeTransitiveClosureReachableFromDefinedPublicConcreteEntries
             throws java.lang.Exception {
  for (Iterator i = definedCode.values().iterator();  i.hasNext();  )  {
        CodeBlob cb = (CodeBlob) i.next();
        if (cb != null   &&  cb.accuracy == AnalysisLevel.CompileTimeType  &&
            cb.isPublic  &&  cb.isConcrete())
            cb.upgradeToRTTAnalysis();
        if (cb.calledCode != null)
            cb.recursiveDescentUpgradation();
  }
};
public void recursiveDescentUpgradation () throws java.lang.Exception {
    for (Iterator j = calledCode.entrySet().iterator();  j.hasNext();  )  {
        Map.Entry a = (Map.Entry) j.next();
        CodeBlob  b = (CodeBlob)  a.getValue();
        if (b != null  &&  b.accuracy != AnalysisLevel.RunTimeType)  {
            b.upgradeToRTTAnalysis();
            b.recursiveDescentUpgradation();
        }
    }
}

public static int abstrctIntrfceVarient () { //returns count
    int retval = 0;
    for (Iterator i = tobeDefinedCode.values().iterator();  i.hasNext();  )  {
        ConcurrentHashMap hm = (ConcurrentHashMap) i.next();
        if (hm != null)
```

```
            retval += hm.size();
        }
        return retval;
    }
    public static int synchVarientCount () {
        int retval = 0;
        for (Iterator i = varients.values().iterator();  i.hasNext();  )  {
            CodeBlob cb = (CodeBlob) i.next();
            if (cb != null  &&  cb.lockType != null)
                retval ++;
        }
        return retval;
    }
    ◇
```

Fragment defined by 31a, 34ab, 46c, 48, 53b.
Fragment referenced in 6.

In the method below, the first parameter is an *out* parameter, which contains the list of *synchronized-edges* as computed by this method. The second parameter is a *in-out* parameter which is used as a work-area to keep track of the path traced by this method, so as to avoid ending-up in infinite-loops along cycles in the underlying call-graph. The third parameter is an *in* parameter that is initialised to the `lockType` of the specific `sCode` member that is being investigated. When tracing a call-path starting from one of its *concrete public* invokers until it reaches the specific `sCode` member (that acquires the said lock), this paramerter is passed-on as is; once the said lock has been acquired, its value is passed on as Null, to indicate that the *from* lock has been acquired. The Code then starts looking for the second lock acquired alog the path, before terminating the *walk*. Until the expected lock is acquired it checks to make sure that if any other lock is acquired, then the path being investigated is disqualified.

⟨ *CodeBlob Instance Methods* 55 ⟩ ≡

```
    public void augmentReachableSCode (Set rv, Set workArea,
                                String firstLock, String callStack)
        throws java.lang.Exception  {
        if (accuracy != AnalysisLevel.RunTimeType)
            upgradeToRTTAnalysis();
        if (calledCode == null)
            return;
        int ti = 2;
        for (Iterator j = calledCode.entrySet().iterator();  j.hasNext();  ti = 2)  {
            Map.Entry a = (Map.Entry) j.next();
            CodeBlob  b = (CodeBlob)  a.getValue();
            if (b == null)  continue;
            if (b.accuracy != AnalysisLevel.RunTimeType)  {
                CodeBlob c = b.upgradeToRTTAnalysis();
                if (c != null  &&  b != c)
                    a.setValue (b = c);
```

```
            }
            if (firstLock != null  &&  firstLock.length() == 0) {
                if (workArea.add (b.getNodeName()))
                    b.augmentReachableSCode (rv, workArea,
                        b.lockType == null ? firstLock : b.lockType,
                        callStack +"-"+ b.getReportableName((String)a.getKey()));
            } else if (firstLock != null) {
                if (b.lockType != null  &&  !b.lockType.equals(firstLock)) {
                    if (ti > 0)  ti--;  else  continue;
                } else if (workArea.add (b.getNodeName()))
                    b.augmentReachableSCode (rv, workArea,
                        (b.lockType!=null && b.lockType.equals(firstLock)) ?  null : firs
                        callStack +"-"+ b.getReportableName((String)a.getKey()));
            }  else  {
                if (b.lockType != null  &&  !b.lockType.equals ("java.lang.Object"))
                    rv.add (callStack +"-"+ b.lockType +"-"+ b.getReportableName(null));
                else if (workArea.add (b.getNodeName()))
                    b.augmentReachableSCode (rv, workArea, null, callStack +"-"+
                        b.getReportableName((String)a.getKey()));
            }
        }
    }
    public String getReportableName(String key) {
        return (getLockType() != null ? ("<" + getLockType() + ">") : "") +
               getNodeName() + (rtReturnTypes == null ? "" : "=M=") +
               (key!=null && !getNodeName().equals(key) && behaviourSpec!=null &&
                key.equals(behaviourSpec.getNodeName()) ? ("=THROandVarientOf="+key) :
                (((key==null || getNodeName().equals(key)) ? "" : ("=THRO=" + key)) +
                 ((behaviourSpec == null) ? "" : ("=VarientOf=" + behaviourSpec.getNodeNa
    };
    ◇
```

Fragment defined by 27a, 32b, 36b, 40, 41, 47, 50, 55.
Fragment referenced in 6.

In the two methods defined above, the `workArea` parameter is used to ensure that the code does not get into loops that involve re-visiting either the same `CodeBlob` along the path being explored. This ensures that the edge-set computation is finite.

Step 6: Complete constructing the TLOG

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 TLOG, and record them into `tlogEdges`.

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

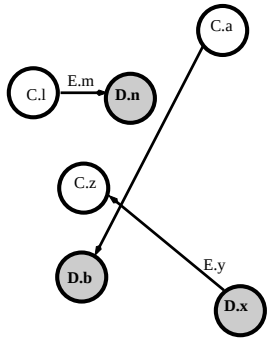

Figure 1.4: The graph after executing Step 4.

collectively all the objects of type `tN`. Each directed-edge in the TLOG 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 TLOG, as constructed by Step 5 from the graph in Figure 4.

⟨ *TLOG Cycle-detection and reporting* 57 ⟩ ≡

```
static public void computeTLOGedges () {
  int rv = 0, nrv = 0;
  for (Iterator i = locks.keySet().iterator();  i.hasNext();  )  {
    String lN  = (String) i.next();                    // lock-Name
    Set    lCS = ((OTlock)locks.get(lN)).lContenders;  // lock-Contender-Set
    if (lCS != null)
        nrv += lCS.size();
    if (lN.charAt(0) == '[')  continue;
    for (Iterator j = lCS.iterator();  j.hasNext();  )  {
      CodeBlob lC = (CodeBlob) j.next();   // lock-Contender
      Set    eSlC = lC.sEdges; // edges-Starting from lock-Conteder
      if (eSlC == null) continue;
      for (Iterator k = eSlC.iterator();  k.hasNext();  )  {
        String eAnnot   = (String) k.next();
        int lastHyphen  = eAnnot.lastIndexOf('-');
        if (lastHyphen >= 0)  {
          // System.err.println ("Before eAnnot = " + eAnnot);
          String tN = eAnnot.substring(lastHyphen+1);  //last method
              assert (tN.startsWith("<"));
              tN = tN.substring (tN.indexOf(">")+1);
              if (tN.indexOf("=") > 0)
                  tN = tN.substring (0, tN.indexOf("="));
              assert (CodeBlob.sCode.get(tN) != null);
          String tLockType = ((CodeBlob)CodeBlob.sCode.get(tN)).getLockType();
          // System.err.println ("After tN = " + tN);
```

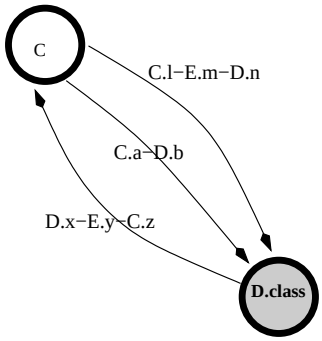

Figure 1.5: The TLOG after executing Step 5.

```
            assert (tLockType != null);
            OTlock  l = (OTlock) OTlock.locks.get(lN);
            HashSet tempM = (HashSet)((HashMap)l.tlogEdges).get(tLockType);
            // Create entry for tLockType into lN.tlogEdges, if not present
            //  the value for this new entry is an object of type HashSet
            if (tempM == null)
              l.tlogEdges.put (tLockType, tempM=new HashSet());
            tempM.add (eAnnot);  // Into this hashSet insert the edgeAnnotation.
            rv++;
          }
        }
      }
    }
    System.out.println ("Pottential TLOG-edges were   = " + nrv);
    System.out.println ("Total TLOG-edges discovered  = " + rv);
  }
  ◇
```

Fragment defined by 57, 58, 62.
Fragment referenced in 8a.

Step 7: Finally, discover and report cycles in the TLOG

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

⟨ *TLOG Cycle-detection and reporting* 58 ⟩ ≡

```
  static public void reportTLOGcycles (int maxLen) {
      assert (maxLen >= 2);
      String  [] cycle      = new String  [maxLen]; // cycle[0]==cycle[n] ==> print
      Iterator[] populator  = new Iterator[maxLen]; // iterative method, not recursive
      for (int len = 2;  len <= maxLen;  len++)  {  // start looking for len-cycles
```

```
            populator[0] = locks.keySet().iterator(); // create iterator to fill cycle[]
            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)  { // select 'legal' next lock
                    cycle[j] = (String) populator[j].next();
                    if (cycle[j].equals ("java.lang.Object"))
                        continue;
                    doSelect = false;
                    doSelect |= ((j>0) && cycle[j].compareTo(cycle[j-1]) <= 0);
                    //for (int k = 0;  k < j;  k++) // If the just populated lock is alre
                    //    doSelect |= (cycle[j].compareTo(cycle[k]) == 0); // the cycle,
                    doSelect |= (j > 0)  &&       // Not a legal selection, so, select ag
                        !((OTlock)locks.get(cycle[j-1])).tlogEdges.keySet().contains(cyc
                    doSelect |= (j == len-1)  &&     // Not a cycle, too bad, select agai
                        !((OTlock)locks.get(cycle[j])).tlogEdges.keySet().contains(cycle
                }
                if (doSelect) {
                    if (j>0 && !populator[j].hasNext())  { // No legal next node; Backtra
                        j--; j--;      // Equivalent to the recursive backtrack;
                    }                  //  '-2' above pre-corrects for the 'j++'
                } else {
                    if (j < len-1)     // If lock-j lock isnt the last lock then
                        populator[j+1] = locks.keySet().iterator(); // reset next iterato
                    if (j == len-1)
                        if (((OTlock)locks.get(cycle[j])).tlogEdges.keySet().contains(cyc
                            printCycle (len, cycle);  // having selected all locks in the
                            for (int k = j;  k>=0;  k--)
                                if (populator[k].hasNext())  {
                                    j = k-1; // '-1' precorrects for j++;
                                    break;   // print the cycle and continue looking
                                }
                        }
                }
            }
        }
        System.out.println ("Total Cycles from all Predictions: " + totalCycles);
    }

    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.hasNext(
                if (((CodeBlob)j.next()).getNodeName().indexOf('#') < 0)  lam++;
                else                                                      lab++;
```

```
            System.out.println ("YYY: " + k + ": " + lam + ":" + lab);
            clam += lam;  clab += lab;
        }
        System.out.println ("Cumulative clam = " + clam + ", clab = " + clab +
                                    ", Total = " + (clam+clab) + ".");
    }
    ◇
```

Fragment defined by 57, 58, 62.
Fragment referenced in 8a.

For our example, the algorithm would print an XML-format report as reproduced below. The report renders the information in the TLOG 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* 60 ⟩ ≡

```
    java.math.BigInteger cumulativeHashValue = java.math.BigInteger.ZERO.abs();
    java.math.BigInteger multiplicativeHashValue = java.math.BigInteger.ONE.abs();
    String ver = ((deadlockPrediction.OTlock)locks.get(cycle[0])).version;
    boolean sameVer = true;
    for (int i = 1;  sameVer  &&  i < len;  i++)
        sameVer = ver.equals(((deadlockPrediction.OTlock)locks.get(cycle[i])).version);
    for (int i = 0;  i < len;  i++) {
        lockNameList += (" " + cycle[i]);
        cumulativeHashValue = cumulativeHashValue.add (java.math.BigInteger.valueOf ((lo
        multiplicativeHashValue = multiplicativeHashValue.multiply(
                                  java.math.BigInteger.valueOf ((long)cycle[i].hashCode(
        if (! sameVer)
            lockNameList += "(" + ((deadlockPrediction.OTlock)locks.get(cycle[i])).versi
```

```
}
lockNameList += "(" + (sameVer ? ver : "Update") + ")";
//long cumulativeHashValue = lockNameList.hashCode();
//if (! alreadyPrintedCumCycles.add(cumulativeHashValue.toString()) &&
//    ! alreadyPrintedMultCycles.add(multiplicativeHashValue.toString()))
//    return;
◇
```

Fragment referenced in 62.

In the cycle printed by the method, we would also like to print information related to the attributes of types whose instance-locks are part of the tlog. 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 `asInstanceAttributes`, all descriptors `type.name`, where the class `type` contains an (non-static) instance attribute `name` of type `K`. Another attribute `asClassAttributes` collects all descriptors `type.sName`, where the class `type` contains a `static` (class) attribute `sName` of type `K`.

⟨ *ClassInfo Instance Attributes* 61a ⟩ ≡

```
public Set asInstanceAttributes = new HashSet();
public Set     asClassAttributes = new HashSet();
public Map    instanceAttributes = new HashMap();
public Map       classAttributes = new HashMap();
public String[] interfacesImplemented = null;
◇
```

Fragment defined by 52a, 61a.
Fragment referenced in 8b.

⟨ *Print list of Attributes associated with instance-locks* 61b ⟩ ≡

```
for (int i = 0;  i < len;  i++)  {
    if (cycle[i].endsWith(".class"))
        continue;
    ⟨ Print list of Attributes (61c asInstance ) 61e ⟩
    ⟨ Print list of Attributes (61d asClass ) 61e ⟩
}
◇
```

Fragment never referenced.

⟨ *Print list of Attributes* 61e ⟩ ≡

```
if (((ClassInfo)CodeBlob.classes.get(cycle[i])).@1Attributes.size() > 0)  {
    System.out.println ("<@1-Attributes-of--" +cycle[i]+ ">");
    for (Iterator j = ((ClassInfo)CodeBlob.classes.get(cycle[i])).
                @1Attributes.iterator();  j.hasNext();  )
        System.out.println ("    " + (String)j.next());
    System.out.println ("</@1-Attributes-of--" +cycle[i]+ ">");
}
◇
```

Fragment referenced in 61b.

⟨ *TLOG Cycle-detection and reporting* 62 ⟩ ≡

```
static private Set alreadyPrintedCumCycles = new HashSet();
static private Set alreadyPrintedMultCycles = new HashSet();
static public int  totalCycles = 0;
static public void printCycle (int len, String[] cycle) {
    String lockNameList = "";
    int cycles = 1;
    ⟨ Store cycle-signature and check to avoid repeats 60 ⟩
    System.err.println ("<Cycle-" +len+ " locks=\""+lockNameList +"\">");
    for (int i = 0;  i < len;  i++)  {
        HashSet stacks = ((HashSet)
                          ((deadlockPrediction.OTlock)locks.get(cycle[i])
                          ).tlogEdges.get(cycle[(i+1)%len]));
        assert (stacks.size() > 0);
        cycles *= stacks.size();
        for (Iterator j = stacks.iterator();  j.hasNext();  ) {
            System.err.println ("<Thread-" + (i+1) + ">");
            System.err.println (((String)j.next()).replace('-', '\n').
                                replaceAll ("\"<init>\"", "\"init\""));
            System.err.println ("</Thread-" + (i+1) + ">");
        }
    }
    System.err.println ("</Cycle-" + len + ">");
    totalCycles += cycles;
}
◇
```

Fragment defined by 57, 58, 62.
Fragment referenced in 8a.

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* 63a ⟩ ≡

```
public ClassInfo (String myNm, String pNm) {
    typeName = myNm;    parentType = pNm;
};
```

◇

Fragment referenced in 8b.

⟨ *ClassInfo Instance Methods* 63b ⟩ ≡

```
public void setAsInstanceAttribute (String nm) {
    asInstanceAttributes.add (nm);
};

public void setAsClassAttribute (String nm) {
    if (asInstanceAttributes.remove (nm))
        asClassAttributes.add (nm);
};
public void setInstanceAttribute (String nm, String type) {
    instanceAttributes.put (nm, type);
};

public void setClassAttribute (String nm, String type) {
    if (instanceAttributes.remove (nm) != null)
        classAttributes.put (nm, type);
};

public void noteInterfaces (String[] implemented) {
    interfacesImplemented = implemented;
}
public List getInterfacesImplemented () {
    List rv = new java.util.ArrayList ();
    for (ClassInfo c = this;  c != null;
         c = c.parentType!=null ? ((ClassInfo)CodeBlob.classes.get(c.parentType)) : n
        if (c.interfacesImplemented != null)
            for (int j = 0;  j < c.interfacesImplemented.length;  j++)
                rv.add (c.interfacesImplemented[j]);
    for (int i = 0;  i < rv.size();  i++)  {
        String   ii = (String) rv.get(i);
```

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.

```
        String[] ie = null;
        if ((ie = (String[]) CodeBlob.interfaces.get(ii)) != null)
            for (int j = 0;  j < ie.length;  j++)
                rv.add (ie[j]);
    }
    return rv;
}
◇
```

Fragment referenced in 8b.

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.

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.
- 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

```
<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/Class;
)Ljava/util/Map;
    sun/reflect/annota...tionParser.parseAnnotations2:([BLsun/reflect/ConstantPool;Ljava/lang/Class;
    sun/reflect/annotation/AnnotationType.getInstance:(Ljava/lang/Class;)Lsun/reflect/annotation/Ann
  <\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/Ann
    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.

`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 code-bases 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, we develop the usage specification for the program, and explore 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 JavaBlob [class-file(s)-in-cwd | a-single-jar-file] ?

We call the program (Class) 'JavaBlob', since each object of this type represents a peice-of-code in the call-graph: 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 the 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:

"JavaBlob.java" 69≡
```
import java.util.*;
import java.util.regex.*;
import java.io.*;
import deadlockPrediction.*;
public class JavaBlob extends deadlockPrediction.CodeBlob {
    ⟨ JavaBlob Instance Attributes 85 ⟩
    ⟨ JavaBlob Constructors 71 ⟩
    ⟨ Globally accessable state for sharing data amongst threads 72a ⟩
    ⟨ Methods to process invocation & invoke javap to create text-rendering 72b, … ⟩
    ⟨ Methods to parse text-rendering and create initial call-graph structures 79a ⟩
```

⟨ *Main method to execute algorithms as per the steps, to print cycles, if any* 70 ⟩
```
}
```
◇

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* 70 ⟩ ≡
```
public static void main (String[] args) {
    try {
        readInputNdCreateCallGraph (args);           // Step 1
        exhaustedReadingInputJars = true;
        inheritFromDefinedCode ();                   // Step 2
        reportSubclassingNdImplements ();            // for the popl paper !!
        //if (true) return;          // No analysis necessary !!
        System.out.println ("sCode cardinality after exhausting input: " + sCode.size
        for (Iterator i = definedCode.values().iterator();  i.hasNext();  )  {
             JavaBlob cb = (JavaBlob) i.next();
             if (cb != null  &&  cb.accuracy != AnalysisLevel.RunTimeType  &&
                 cb.aReturnCount > 0)
                 cb.upgradeToRTTAnalysis();
        }
        for (int fpCount = 1; --fpCount > 0;  )
           for (Iterator i = definedCode.values().iterator();  i.hasNext();  )  {
               JavaBlob cb = (JavaBlob) i.next();
               if (cb != null  &&  cb.accuracy != AnalysisLevel.RunTimeType  &&
                   cb.monitorExits != null)
                   cb.upgradeToRTTAnalysis();
           }
        System.out.println ("sCode cardinality after interpreting non-null monitorexi
        // interpretSynchNdVarients ();                  // Step 4
        // interpretInterfaceVarients ();                // Step 4
          countStartPoints ();                          // Step 3
        // collectStartPoints ();                       // Step 3
        ⟨ Report Statistics 53a ⟩
        System.out.println ("Starting to execute Step 4.");
        for (int fpCount = 3; --fpCount > 0;  )
           for (Iterator i = definedCode.values().iterator();  i.hasNext();  )  {
               JavaBlob cb = (JavaBlob) i.next();
               if (cb != null  &&  cb.accuracy != AnalysisLevel.RunTimeType)
                   cb.upgradeToRTTAnalysis();
           }
        computeTransitiveClosureReachableFromDefinedPublicConcreteEntries ();
        System.out.println ("sCode cardinality after computing Transitive Closure: "
        deadlockPrediction.OTlock.reportLockUsage();
        sCodeReachedFromSCode ();                       // Step 5
```

```
        ⟨ Report Statistics 53a ⟩
        deadlockPrediction.OTlock.computeTLOGedges ();    // Step 6
        deadlockPrediction.OTlock.reportTLOGcycles (4);   // Step 7
        ⟨ Report Statistics 53a ⟩
    } catch (Exception ex) {
        ex.printStackTrace(System.out);
    }
}
/*  System.out.println ("Input Class  count: \t" + classesList.size() + ".");
 *  Set extra = classes.keySet();
 *  classesList.removeAll (interfaces);
 *  extra.removeAll (classesList);
 *  Object[] overDefinedClasses = extra.toArray();
 *  System.out.println ("Over defined Class count:" + overDefinedClasses.length);
 *  for (int k = overDefinedClasses.length;  --k >= 0;  )
 *      System.out.println ("Overdefined Class: " + ((String)overDefinedClasses[k]))
 */
◇
```

Fragment referenced in 69.

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* 71 ⟩ ≡

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

public JavaBlob (MethodSignature ms, boolean isStat,
                 boolean isSynch, boolean isPub)  {
    super (ms.signature(), isStat, isSynch, isPub);
}

public JavaBlob (JavaBlob p, String c, String lT)  {
    super ((deadlockPrediction.CodeBlob) p, c, lT);
}
public JavaBlob (JavaBlob ct, String rt)  {
    super ((deadlockPrediction.CodeBlob) ct, rt);
}
◇
```

Fragment referenced in 69.

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. The *javap invoker thread* populates the `textIn` variable below, before seting the `goConsumerGo` variable to true. The main thread awaits the (control-variable) `goCosumerGo` variable to become `true` before accessing the input from `textIn`. As soon as it is done using all the input fron `textIn`, it resets `goConsumerGo` to `false`, indicating to the other thread that it is now ready for the next bunch of input.

⟨ *Globally accessable state for sharing data amongst threads* 72a ⟩ ≡

```
static BufferedReader textIn = null;
static volatile java.util.concurrent.SynchronousQueue<BufferedReader> javaPout =
       new java.util.concurrent.SynchronousQueue<BufferedReader>();
static String         classesHome = null;
static Set            classesList = null;
◇
```

Fragment referenced in 69.

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* 72b ⟩ ≡

```
static public void classListGivenUsage(String[] args)  {
    System.out.println ("Argument.length = " + args.length);
    if (args.length==0)
        classesList = classNamesFromJar (null);
    else if (args.length==2) {
        assert (args[args.length-2].indexOf(".txt")>=0);
        try {
        BufferedReader in = new BufferedReader (
                            new FileReader (args[args.length-2]));
        classesList = new TreeSet();
        for (String line = in.readLine();  line != null;  line = in.readLine())  {
            //line = line.trim().replace ('/', '.');
            //line = line.substring (0, line.lastIndexOf(".class"));
            if (!classesList.add(line))
                System.err.println ("Dropped here: " + line);
        }   System.out.println ("ClassList cardinality: " + classesList.size());
        } catch (java.lang.Exception e) {
            System.out.println ("Exception: " + e);
        }
        classesHome = args[args.length-1];
```

```
        } 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 72b, 73, 75, 76.
Fragment referenced in 69.

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* 73 ⟩ ≡

```
    static public Set 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") || version.startsWith ("1.6"))
                    jar += ' ' + home + "/lib/ext/sunjce_provider.jar " +
                                 home + "/lib/ext/dnsns.jar " +
                                 home + "/lib/ext/sunpkcs11.jar " +
                                 home + "/lib/ext/localedata.jar";
                else if (version.startsWith ("1.4"))
                    jar += ' ' + home + "/lib/sunrsasign.jar " +
                                 home + "/lib/ext/localedata.jar " +
                                 home + "/lib/ext/sunjce_provider.jar " +
                                 home + "/lib/ext/dnsns.jar " +
                                 home + "/lib/ext/ldapsec.jar";
            }

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

```
            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 72b, 73, 75, 76.
Fragment referenced in 69.

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" 74a≡

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

"grepNDcutInLoop" 74b≡

```
    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* 75 ⟩ ≡

```
static final int  NoOfClassesPerJavapInvocation = 1;
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.hasNext();  )  {
                        String t = (String) i.next();
                        subList = subList + " " + t;
                        if (breakpoint (t))  break;
                    }
                    String delete2be = System.getProperty ("java.home") +
                           "/../bin/javap -private -c -verbose -J-Xmx424m " +
                           (classesHome!=null ? ("-classpath "+classesHome) : "")
                           + " " + subList;
                    System.err.println (subList.replace (' ', '\n'));
                    child = Runtime.getRuntime().exec (System.getProperty ("java.home
                                "/../bin/javap -private -c -verbose -J-Xmx424m " +
                        (classesHome!=null ? ("-classpath " + classesHome) : "")
                           + " " + subList);
                    r = new InputStreamReader (child.getInputStream());
                    javaPout.put (new BufferedReader (r));
                    if (child.waitFor() != 0)
                        System.err.println("javap (" + subList + ") Failed: " +
                                           child.exitValue());
                } while (i.hasNext());
            } catch (Exception e) {
                System.out.println (e.toString());
            }
        }
    };
```

```
        new Thread(invoker).start();
    }
    ◇
```

Fragment defined by 72b, 73, 75, 76.
Fragment referenced in 69.

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* 76 ⟩ ≡

```
    public static boolean breakpoint (String fqcn)  {
        String version = System.getProperty ("java.version");
        if (fqcn == null) return true;
        if (fqcn.startsWith ("sun.net.spi") ||
            fqcn.equals ("com.sun.crypto.provider") ||
            fqcn.startsWith ("sun.text.resources"))
            return true;
        if (version.startsWith ("1.7"))
            return  fqcn.startsWith ("com.sun.org.apache") ||
                    fqcn.startsWith ("com.sun.xml.internal.ws")  ||
                    fqcn.startsWith ("java.io")  ||  fqcn.startsWith ("javax.swing")  ||
                    fqcn.startsWith ("javax.xml")  ||  fqcn.startsWith ("java.util.regex
                    fqcn.startsWith ("sun.awt")  ||  fqcn.startsWith ("sun.java2d.cmm.lc
        else if (version.startsWith ("1.6"))
            return  fqcn.startsWith ("com.sun.corba.se.spi.transport") ||
                    fqcn.startsWith ("sun.util.resources") ||
                    fqcn.startsWith ("com.sun.crypto.provider.TlsRsaPremasterSecretGener
                    fqcn.startsWith ("com.sun.crypto.provider.ai") ||
                    fqcn.startsWith ("sun.security.pkcs11.wrapper.PKCS11RuntimeException
        else if (version.startsWith ("1.5"))
            return  fqcn.startsWith ("sun.security.pkcs11.wrapper.PKCS11RuntimeException
                    fqcn.startsWith ("sun.net.www");
        else if (version.startsWith ("1.4"))
            return  fqcn.startsWith ("com.sun.jndi.ldap") ||
                    fqcn.startsWith ("sun.net.www") ||
                    fqcn.startsWith ("com.sun.security.sasl.util.SaslImpl");
        return true;
    }
    ◇
```

Fragment defined by 72b, 73, 75, 76.
Fragment referenced in 69.

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* 77a ⟩ ≡

```
import java.io;
package mlTest;
class MLtest  {
  class NmlTest {
    static synchronized void main (string[] args) {
      System.out.println ("Max Integer: " + 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* 77b ⟩ ≡

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

public static synchronized void main(java.lang.String[]);
  Code:
   0:   getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc     #3; //String The Max Integer value: 2147483647
```

```
        5:   invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;
        8:   return

    }◇
```
Fragment never referenced.

Main loop to parse text-rendering and create call-graph

The input stream `textIn`, generated as the *verbose* output of `javap` contains a sequence of class-definitions (like in the example above), interspersed by the occassional interface-definition. The main loop reads input from the class-variable `textIn`, and identifies the points where the the class-header, and interface-header begin, and call their processsing methods respectively. 'class' definitions begin with the word "class", and 'interface' definitions begin with the word "interface". They are to be found on the top-level, not nested within any-other.

⟨ *Main loop to parse text-rendering* 78 ⟩ ≡
```
    public static void
    readInputNdCreateCallGraph (String[] args) throws java.lang.Exception {
        if ((textIn = new BufferedReader (new FileReader ("inputFromFile/1.6.classConten
          String line;
          while ((line=textIn.readLine()) != null)
            if (line.indexOf('#')>=0 || line.indexOf('=')>=0)
                continue;
            else if (line.trim().startsWith ("Classfile jar:")) { //1.7.XXX Specific Req
              assert (textIn.readLine().trim().startsWith("Last modified"));
              assert (textIn.readLine().trim().startsWith("MD5 checksum"));
            } else if (line.indexOf("interface ")>=0)
              readInterface (line);
            else if (line.indexOf("class ")>=0)
              readClassNameProcessBody  (line);
            else if (line.trim().length() > 0  &&  !line.trim().startsWith("Compiled fro
              System.err.println ("WHAT IS THIS: " + line);
              assert (false);
            }
        }
    }
    ◇
```
Fragment referenced in 79a.

The code for this section is organised as a collection of methods, each processing a part of the input, and all of which are invoked through a main-loop that identifies the boundaries between them. The main loop returns when all the input has been processed, the core part of the call-graph is already built, and populated. The "Method Body Processors" from the following list are detailed in the next sub-section.

⟨ *Methods to parse text-rendering and create initial call-graph structures* 79a ⟩ ≡
⟨ *Interface declaration Processor* 79b ⟩
⟨ *Class definition Processor* 80 ⟩
⟨ *Method Header Processor* 82 ⟩
⟨ *Field Name & Type Processor* 84a ⟩
⟨ *Method Body Processors* 86, … ⟩
⟨ *Main loop to parse text-rendering* 78 ⟩
◇

Fragment referenced in 69.

Recording the Interface-name

The method defined below records the name of the interface encountered during the code-parse, and skips the rest of the body. The body of the interface is skipped since it cannot define any method-behaviour.

⟨ *Interface declaration Processor* 79b ⟩ ≡

```
static void readInterface (String line)
    throws java.lang.Exception {
    String iii = "", iNm = line.substring (line.indexOf(" interface ") +
                                           " interface ".length());
    System.out.println ("Starting to read INTERFACE: " + iNm);
    if (iNm.indexOf(' ') > 0)
        iNm  = iNm.substring (0, iNm.indexOf(' '));
    if (iNm.indexOf('<') > 0)
        iNm  = iNm.substring (0, iNm.indexOf('<'));
    int indExtends = line.indexOf(" extends ");
    assert (iNm.indexOf('<') < 0);
    if (indExtends >= 0) {
        String ii = line.substring(indExtends + " extends ".length());
        for (int i = 0, j = 0;  i < ii.length();  i++)  {
            if (ii.charAt(i) == '<')           j++;
            if (j == 0)                        iii += ii.charAt(i);
            if (j>0  &&  ii.charAt(i) == '>')  j--;
        }
    }
    interfaces.put (iNm, (indExtends < 0) ? null : iii.split(","));
    while (! (line=textIn.readLine().trim()).equals  ("{"));
    processInterfaceMethodList(iNm);
    System.err.println (iNm + " -- DONE");
};
static void processInterfaceMethodList (String iNm)
    throws java.lang.Exception {
    String line = null;
    while ((line=textIn.readLine()) != null)  {
        line = line.trim();
        if (line.endsWith("}"))     break;
```

```
            assert (!line.startsWith("Compiled from "));
            if (line.length() == 0)     continue;
            if (line.startsWith("Exceptions:")) {
                line = textIn.readLine();
                continue;
            }
            int openP  = line.indexOf("("),
                closeP = line.indexOf(")");
            if (line.indexOf('=')<0 && line.indexOf('#')<0 &&
                line.indexOf(':')<0 && openP>=0 && closeP>=0 && openP<closeP) {
                assert (line.substring(closeP+1).trim().startsWith(";")  ||
                        line.substring(closeP+1).trim().startsWith("throws "));
                String mNm = readMethodHeader(iNm, line);
                assert (mNm == null);
            }
        }
    };
    ◇
```

Fragment referenced in 79a.

Processing a Class-definition

A class definition comprises of the class-name, followed by details of the constants-array, followed by an open-brace, member(s), and method definitions until the class-close is flagged by a close-brace on a line all by itself. After recording the class-name form the opening line, we also register the names and types of all the (non-basic-type) attributes. Once the body of the class begins (flagged by the open-brace), we are interested in the method-headers and the corresponding body-definitions, which we process through two seperate methods.

The Method-definitions have an open and close parenthesis on the same line as their descriptor, followed by the method-body. Detecting the termination of this method-body is slightly tricky. It terminates differently based upon whether the method throws any exceptions. The class-constructor-common class-static initializer starts with a ”static ” on a line by itself. These ’signatures’ are recongnised and control transferred appropriately, below:

⟨ *Class definition Processor* 80 ⟩ ≡

```
    public static void readClassNameProcessBody (String line)
        throws java.lang.Exception {
        String cNm = readClassName(line);
        processFieldNmNdType (cNm);
        System.out.println ("Starting to read CLASS: " + cNm);
        while ((line=textIn.readLine()) != null)  {
            line = line.trim();
            assert (!line.startsWith ("Code:")  &&
                    !line.toLowerCase().startsWith ("stack:"));
```

```
            int openP, closeP;
            if (line.length() == 0)       continue;
            assert (!line.startsWith("Compiled from "));
            if (line.startsWith("}"))     break; // reached end-of-class.
            openP  = line.indexOf("(");
            closeP = line.indexOf(")");
            if (line.indexOf('=')<0 && line.indexOf('#')<0 &&
                line.indexOf(':')<0 && openP>=0 && closeP>=0 && openP<closeP) {
                assert (line.substring(closeP+1).trim().startsWith(";")  ||
                        line.substring(closeP+1).trim().startsWith("throws "));
                String mNm = readMethodHeader(cNm, line);
                if (mNm != null) {
                    ((JavaBlob)definedCode.get(mNm)).processMethodBody();
                    if (line.indexOf("   throws ") > 0) {
                        textIn.mark(256);
                        line = textIn.readLine();
                        if (! line.trim().startsWith("throws "))
                            textIn.reset();
                    }
                }
            }  else if (line.trim().startsWith("static {};"))
                while ((line = textIn.readLine()) != null  &&
                       line.trim().length() > 0  &&
                       !line.trim().startsWith("LineNumberTable:"));

        }
        System.err.println (cNm + " -- DONE");
        // System.out.println ("Completed reading CLASS: " + cNm);
    };

    public static String readClassName (String line) {
        assert (line.indexOf("class ")>=0);
        boolean isAbstract = line.indexOf("abstract ") >= 0;
        int     indExtends = line.indexOf(" extends ");
        String  cNm = null, pNm = null;
        if (indExtends >= 0)  {
            cNm  = line.substring (line.indexOf(" class ")+7, indExtends);
            line = line.substring (indExtends + " extends ".length());
            pNm  = line.indexOf(' ')>0 ? line.substring(0, line.indexOf(' ')) : line;
            pNm  = pNm.replace('/', '.').intern();
        } else {
            line = line.substring (line.indexOf(" class ")+7);
            cNm  = line.indexOf(' ')>0 ? line.substring(0, line.indexOf(' ')) : line;
        }
        cNm = cNm.replace('/', '.').intern();
        if (cNm.indexOf('<') >= 0)
            cNm = cNm.substring (0, cNm.indexOf('<'));
        assert (cNm.indexOf('<') < 0);
        ClassInfo tv  = (ClassInfo) classes.get (cNm);
        if (tv == null) {
```

```
            classes.put (cNm, (tv = new ClassInfo (cNm, pNm)));
            System.out.println ("added into classes: " + cNm + ":" + pNm);
        } else            tv.parentType = pNm;
        tv.isAbstract = isAbstract;
        int indImplements = line.indexOf(" implements ");
        if (indImplements >= 0) {
            String iii = "", ii = line.substring(indImplements + " implements ".length()
            for (int i = 0, j = 0;  i < ii.length();  i++)  {
                if (ii.charAt(i) == '<')           j++;
                if (j == 0)                        iii += ii.charAt(i);
                if (j>0  &&  ii.charAt(i) == '>')  j--;
            }
            tv.noteInterfaces (iii.split(","));
        }
        return cNm;
    }
    ◇
```

Fragment referenced in 79a.

Method Header Processing

The method below processes the Method-header. It returns a non-null method-name whenever its body is expected, and to be processed. For Native, and abstract methods, it returns null. Interface-method declarations are bypassed, not processed at all.

⟨ *Method Header Processor* 82 ⟩ ≡

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

        int lastBlankB4OpenP = line.substring(0, openP).lastIndexOf(" ");
        mNm = line.substring(lastBlankB4OpenP+1, openP);
        if (lastBlankB4OpenP >= 0)
            decoration = line.substring(0, lastBlankB4OpenP).trim();
        if (decoration.endsWith(">")) {
            assert (decoration.indexOf ("<") >= 0);
            decoration = decoration.substring (0, decoration.indexOf("<"));
        }

        // String[] pTypes   = line.substring(openP+1, closeP).split(",");
        if (cName.replace('/', '.').equals(mNm))
            mNm = "\"<init>\"";  // Encountered Constructor. ==> rType is null.
        else  {
            int lastBlankB4MethodName = decoration.lastIndexOf (" ");
            if (lastBlankB4MethodName < 0)  {
                rType = decoration;
```

```
                decoration = "";
            }  else  {
                rType = decoration.substring (lastBlankB4MethodName + 1);
                decoration = decoration.substring (0, lastBlankB4MethodName);
            }
        }
        // System.out.println ("Decoration: " + decoration);
        MethodSignature test = new MethodSignature (cName, mNm, // prt = getPRTcode (pTy
                               line.substring(openP, closeP+1), new TypeSignature (rType
        JavaBlob tv = (JavaBlob) definedCode.get (test.signature());
        boolean isNative   = decoration.indexOf("native")    >= 0;
        boolean isAbstract = decoration.indexOf("abstract")  >= 0;
        if (isNative || isAbstract) {
            nativeOrAbstract (test.signature(), isNative);
            return null;
        } else {
            boolean isPublic   = decoration.indexOf("public")    >= 0;
            boolean isStatic   = decoration.indexOf("static")    >= 0;
            boolean isSynch = decoration.indexOf("synchronized") >= 0;
            if (tv == null)
                new JavaBlob (test, isStatic, isSynch, isPublic);
            else
                tv.upgradeCB (isStatic, isSynch, isPublic);
            return test.signature();
        }
    }
    ◇
```

Fragment referenced in 79a.

Field Name & Type processing

The constant-table details printed after the class-header, and before the corresponding open-brace contains (among other details) entries corresponding to the various fields of the class, along with their respective type-name. For instance, the relevant infomation can look like:

```
const #353 = Field      #200.#267;      //  java/lang/String.count:I
const #354 = Field      #200.#268;      //  java/lang/String.hash:I
const #355 = Field      #200.#269;      //  java/lang/String.offset:I
```

These details are fetched from the text-rendering of the java.lang.String class, and the first line indicates that there is a field called `count` of type `int`, in that class. The one thing that cannot be properly deciphered from the above-like fragment is as to whether the field is an instance-field or a (static) class-attribute. In order to detect that, our analysis firstly registers all ”fields” (as instance-fields) from the above-like table, and subsequently, based upon the byte-code used to access it (which can be either

`get/putfield` or `get/putstatic`) re-register the static fields as being class-attributes. The first part is done by the function below, and the second part can only be done by the method `processMethodBody`, defined later, below.

⟨ *Field Name & Type Processor* 84a ⟩ ≡

```
static void processFieldNmNdType (String cNm)
    throws java.lang.Exception {
    String line = null;
    while ((line=textIn.readLine()) != null)  {
        if ((line = line.trim()).equals("{"))
            break;
        if ((line.startsWith("const #") || line.trim().startsWith("#")) &&
            line.indexOf("= Field")>0) {
            ⟨ Pick fType and retain attribute-name (84b "//   ") 84c ⟩
            /* ClassInfo tv = (ClassInfo) classes.get (fType.name());
            if (tv == null)
                classes.put (fType.name(), tv = new ClassInfo (fType.name(), null));
            tv.setAsInstanceAttribute (line);
            tv = (ClassInfo) classes.get (cNm);
            if (! TypeSignature.basicEncoded.contains (fType.signature()))  {
                tv.setInstanceAttribute (line.substring(line.lastIndexOf('.')+1),
                                         fType.signature());
                // System.out.println ("Added <" + fType.signature() + ", " + line +
            } */
        }
    }
}
◇
```

Fragment referenced in 79a.

⟨ *Pick fType and retain attribute-name* 84c ⟩ ≡

```
if (! line.endsWith(";"))
    continue;
TypeSignature fType = new TypeSignature (line.substring (line.indexOf(':')+1), true)
line = line.substring (line.indexOf(@1) + @1.length(),
                       line.indexOf(':')).replace('/', '.');
// System.out.println ("Read <" + fType.signature() + " / " + fType.name() + " : "+
◇
```

Fragment referenced in 84a, 86, 94.

1.8.3 Processing the method body

Recollect from the discussion in the previous section describing the cycle-detection algorithm, that our OO-Analysis requires an abstract-interpretation of the method-body at the level of types. In the interest of simplicity we organise our implementation in the form of a set of methods that are described, and detailed below:

1. Before we proceed to process the specification 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 "Exceptions:", or 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 details under "Exceptions:" and the "LineNumberTable:" are 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. 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.

5. In the first pass, we read the text-rendering off the input-stream, and create two 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` $>$. The subsequent pass(es) use the data stored in the arrays. Besides these, there are another array called `monitorExits`: which is populated using the contents of the "Exception Table:", if present, by another method: `readExceptionNdLineNoTable`.

⟨ *JavaBlob Instance Attributes* 85 ⟩ ≡

```
public int      locals = -1;
public int[]    offset = null;
public String[] code   = null;
public List     monitorExits = null;
public int      aReturnCount = 0;
```

◇

Fragment referenced in 69.

⟨ *Method Body Processors* 86 ⟩ ≡

```
public void processMethodBody() throws java.lang.Exception {
    String version = System.getProperty ("java.version");
    String line = textIn.readLine(), tl = null;
    boolean monitorUsed = false;
    assert (getContainerType().name() != null);
    System.out.println ("Started reading method: " + getNodeName());
    if (line.trim().startsWith("flags:"))
        line = textIn.readLine();
    if (line.trim().startsWith("Synthetic:"))
        line = textIn.readLine();
    assert (line.trim().startsWith("Code:"));
    assert ((line = textIn.readLine()) != null  &&  line.indexOf("ocals=") > 0);
    locals = Integer.parseInt (tl = line.substring (
        line.indexOf("ocals=")+"ocals=".length(), line.toLowerCase().indexOf(", args_
    totalLocals += locals;
    int bcOffset, colon = -1;
    List toffset = new ArrayList(),  tcode = new ArrayList();

    while ((line=textIn.readLine()) != null  &&  line.trim().length() > 0)  {
        if (line.trim().startsWith("Exceptions:")  ||// point 2 above.
            line.trim().startsWith("}"))            // Surprise!: reached end-of-cla
            break;
        else if (line.trim().startsWith("Exception table:"))  {  // point 1 above.
            readExceptionNdLineNoTables (toffset, tcode);
            break;
        } else if (line.trim().startsWith("LineNumberTable:")  ||
                   line.trim().startsWith("StackMapTable:")) {
            while ((line = textIn.readLine()) != null  &&
                   line.trim().length() > 0  &&  !line.trim().equals("}"));
            break;
        }
        if ((colon = line.indexOf(':')) < 0) {
            System.out.println ("How-come this is read here: " + line);
            assert (false);
        }
        bcOffset = Integer.parseInt (line.substring(0, colon).trim());
        if (line.indexOf("tableswitch{")>=0  || line.indexOf("tableswitch   {")>=0
            line.indexOf("lookupswitch{")>=0 || line.indexOf("lookupswitch  {")>=0)
            do {
                line = line + (tl = textIn.readLine());   // Multi-line instructions
            } while ((version.startsWith("1.6") &&  tl.indexOf("default:")<0)  ||
                     (version.startsWith("1.7") && !tl.trim().equals("}")));
        line = addDefiningContextIfNecessary (line.substring(colon+1).trim());
        toffset.add (bcOffset);
```

```
            tcode.add   (line);
            monitorUsed = monitorUsed || line.startsWith("monitore");  // point 6 above
            if (line.startsWith("areturn"))                aReturnCount++;
                 if (newOp  .matcher(line).lookingAt())         newCount++;
            else if (stores .matcher(line).lookingAt())        ssaLocals++;
            else if (invokes.matcher(line).lookingAt()) invocationsCount++;
            if ((getLockType()==null && !monitorUsed)  &&
                (line.startsWith("invokevirtual\t")   ||    // point 5 above
                 line.startsWith("invokespecial\t")   ||    // -- " --
                 line.startsWith("invokeinterface\t") ||    // -- " --
                 line.startsWith("invokestatic\t"))) {      // -- " --
                String ms = line.substring (line.indexOf("; //") + "; //".length());
                       ms = line.substring (line.indexOf("Method ") + "Method ".length()
                //interpretInvocation (ms, line.startsWith("invokestatic\t"),
                //                         line.startsWith("invokespecial\t"),
                //                         line.startsWith("invokeinterface\t"),
                //                         line.startsWith("invokevirtual\t"), null);
            } else if (line.indexOf("tstatic")==2 &&  //line.indexOf("//Field ")>0) {
                       line.substring(line.indexOf("//")+2).trim().startsWith("Field "))
                line = line.substring(line.indexOf("//")+2).trim();
                ⟨ Pick fType and retain attribute-name (87 "Field " ) 84c ⟩
                if (line.indexOf('.') <= 0)
                    line = getContainerType().name() + '.' + line;
                /* ClassInfo tv = (ClassInfo) classes.get (fType.name());
                assert (tv != null);
                tv.setAsClassAttribute (line);
                assert ((tv = (ClassInfo) classes.get (getContainerType().name())) != nu
                if (! TypeSignature.basicEncoded.contains (fType.signature())) {
                    tv.setClassAttribute (line.substring(line.lastIndexOf('.')+1),
                                          fType.signature());
                    // System.out.println ("Re-Setting " +
                    //       line.substring(line.lastIndexOf('.')+1) + ":" + fType.signa
                }*/
            }
        }
        offset = new int [toffset.size()];
        code   = new String [tcode.size()];
        for (int j = toffset.size();  --j >= 0;  ) {
            offset[j] = ((Integer) toffset.get(j)).intValue();
            code[j]   = (String) tcode.get(j);
        }
        toffset.clear();  tcode.clear();
        accuracy = AnalysisLevel.CompileTimeType;
        System.out.println ("Completed reading method: " + getNodeName());
        assert (!monitorUsed || monitorExits.size()>0);
    }

    public String addDefiningContextIfNecessary (String l) {
        String rv = l;
        if (l.startsWith("invokevirtual\t")   ||
```

```
            l.startsWith("invokespecial\t")   ||
            l.startsWith("invokeinterface\t") ||
            l.startsWith("invokestatic\t")) {
            String ms = l.substring (l.indexOf("; //") + "; //".length());
                   ms = l.substring (l.indexOf("Method ") + "Method ".length());
            if (! MethodSignature.isFullySpecified (ms))
                rv = l.substring (0, l.indexOf(ms)) + getContainerType().name() + "." + r
            //if (! l.equals(rv))
            //    System.out.println ("addDefiningContextIfNecessary: " + l + "/" + rv);
        }
        return rv;
    };
    ◇
```

Fragment defined by 86, 91, 94, 98b, 106.
Fragment referenced in 79a.

Finding the matching *monitorexit*

The contents of the "Exception Table:" can be used to find the *matching* `monitorexit` instruction for every `monitorenter` instruction. Below, through a running example, we show how this can be accomplished. 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
```

1. 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 occurs during the attempt to

compute `this.doThat()`. So also, notice that the ”Exception table:” has two entries that have ’25’ as the target, and ’any’ as their ’Type’. The first of them encapsulates the entire body of the `synchronized` statement between its ’from’ and ’to’ 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-processing’ of 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#.

2. 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:
 ...
```

3. From the above table, we firstly discount all entries that have the same value for the ’From’ and the ’Target’ column, while having ’any’ in the ’Type’ column.

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:
 ...
```

4. The next step of processing merely registers (into `execStartOffsets`) 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:
 ...
```

5. 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
```

⟨ *Method Body Processors* 91 ⟩ ≡

```
public void readExceptionNdLineNoTables(List offset, List code)
    throws java.lang.Exception {
    String version = System.getProperty ("java.version");
    assert (textIn.readLine().trim().startsWith("from"));
    int    i[] = new int[50], j[] = new int[50], k[] = new int[50],
           t=-1, n=0, ni=0, nj=0, nk=0;
    String l = textIn.readLine();
    for (;  l != null  &&  l.trim().length()>0  &&
            !l.trim().startsWith("LineNumberTable:")  &&
            !l.trim().startsWith("StackMapTable:")  &&
            !l.trim().startsWith("Exceptions:");  l = textIn.readLine())  {
        ni = Integer.parseInt ((l = l.trim()).substring (0, t = l.indexOf(' ')));
        l  = l.substring(t).trim(); // read and drop the 'from' int value
        nj = Integer.parseInt (l.substring (0, t = l.indexOf(' ')));
        l  = l.substring(t).trim(); // read and drop the 'to' int value
        nk = Integer.parseInt (l.substring (0, t = l.indexOf(' ')));
        l  = l.substring(t).trim(); // read and drop the 'target' int value
        if (l.startsWith("any")  &&  ni != nk  &&
            ((String)code.get(offset.indexOf(nj)-1)).startsWith("monitore"))  {
               i[n] = ni;  j[n] = nj;  k[n] = nk;  n++;
               // System.out.println ("Processed: " +ni+','+nj+','+nk+','+n);
        } else if (l.startsWith("Class") && !version.startsWith("1.7")) {
            //System.out.println ("l reads: " + l);
            l = textIn.readLine();
            //System.out.println ("l reads: " + l);
            assert (l==null  ||  l.trim().length()==0);
        }
    }
    monitorExits = new ArrayList();
    for (int m = 0;  m < n;  m++)
        if (m != (n-1)  &&  k[m] == k[m+1]) {
            if ((offset.indexOf(j[m])+1) != offset.indexOf(i[m+1]))
                System.out.println ("Doing inaccurate analysis of " + getNodeName())
            i[m+1] = i[m];
        } else if (i[m] != 0  && ((String)code.get(offset.indexOf(i[m])-1)).
                                          startsWith("monitorenter"))
            monitorExits.add (j[m]);
    // System.out.println ("Matching Exits at n(" + n + "): " + monitorExits);
    if (l!=null && l.trim().length()>0 && l.trim().startsWith("StackMapTable:")) {
        while ((l=textIn.readLine()) != null  &&  l.trim().length() > 0);
        System.out.println ("Read after StackMapTable: " + (l = textIn.readLine()));
    }
    if (l!=null && l.trim().length()>0 && l.trim().startsWith("LineNumberTable:"))
        while ((l=textIn.readLine()) != null  &&
               l.trim().length() > 0  &&  !l.trim().equals("}"));
```

```
}
◇
```
Fragment defined by 86, 91, 94, 98b, 106.
Fragment referenced in 79a.

Abstract interpretation of the method-body

1. There are two tasks, primarily, that need to be accomplished during the second pass: one is to properly process the monitor -enter/-exit byte-codes as is already being done, and another is to record the run-time-type information of objects created, and assigned, and appropriately use this RTTI to accurately record the call-graph data.

2. 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. (Therefore, the length of `typeAtTOSlist` is the same as the length of `offset` array for that method). 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`.

3. 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.

4. As for the monitor-enter/-exit processing, 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.

5. 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.

6. 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*.

7. 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`.

8. 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`.

9. 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: <`execStartOffsets`, `stackAtExecStart`, and `arrayAtExecStart`>. 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.

10. 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`.

11. 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 `execStartOffsets`, 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.

12. 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 (sequentially-)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* 94 ⟩ ≡

```
public static Set bfcSet = new HashSet();
public void upgradeToRTTAnalysis (
    deadlockPrediction.CodeBlob spec)  throws java.lang.Exception {
    if (accuracy == AnalysisLevel.RunTimeType)
        return;
    if (spec == null || bfcSet.contains (spec.getNodeName()))  {
        System.out.println ("XXX No Interpretation for null behavioured: " +getNodeNa
        return;
    };
    if (spec != this) {
```

```
            isPublic  = spec.isPublic;
            isStatic  = spec.isStatic;
            String lt = spec.getLockType();
            if (lt != null  &&  accuracy == AnalysisLevel.Null)
                setLockType (getContainerType().name() +
                             (lt.endsWith(".class") ? ".class" : ""));
            locals = ((JavaBlob) spec).locals;
            offset = ((JavaBlob) spec).offset;
            code   = ((JavaBlob) spec).code;
            monitorExits = ((JavaBlob) spec).monitorExits;
            assert (! getNodeName().equals (spec.getNodeName()));
            behaviourSpec = spec;
        }
        if (offset == null) {
            System.out.println ("Offset null for: " + getNodeName() +
                                "\n      From Spec: " + spec.getNodeName());
            return;
        }
        System.out.println ("Interpreting: " +getNodeName()+
              (spec==this ? "." : ("\n        From: " + spec.getNodeName())));
        rttAnalysisCount++;
        List execStartOffsets = new LinkedList(); // pc(s) where executions (re)start
        List stackAtExecStart = new LinkedList(); // Stack-state at those PC(s)
        List arrayAtExecStart = new LinkedList(); // Local-state at those PC(s)
        java.lang.Object[] typeAtTOSlist = new java.lang.Object[offset.length];
            // Execution begins at PC-Zero, with an empty stack.
        updateBBlist (0, new Stack(), getTypeArrayFromParams (locals),
                      execStartOffsets, stackAtExecStart, arrayAtExecStart);
        boolean doneOffsets[] = new boolean [offset.length];
        Stack objStack = new Stack();   objStack.push (this);
        StringBuffer eCounter = new StringBuffer("0");
        for (int i = 0;  i < execStartOffsets.size();  i++)  {
            int bcOffset       = ((Integer)execStartOffsets.get(i)).intValue();
            Object[] typeArray =  (Object[]) arrayAtExecStart.get(i);
            Stack typeStack =  (Stack) stackAtExecStart.get(i);
            assert (indexOfOffset(bcOffset) != -1);
            for (int j = indexOfOffset(bcOffset);
                 j<offset.length && !doneOffsets[j];  j++)  {
                typeAtTOSlist[j] = typeStack.empty() ? null : typeStack.peek();
                doneOffsets[j] = true;
                // printArrayNdStack (typeArray, typeStack);
                if (! interpret(objStack, j, typeStack, typeArray,
                      execStartOffsets, stackAtExecStart, arrayAtExecStart, eCounter))
                    break;
            }
        }
        accuracy = AnalysisLevel.RunTimeType;
    }

    public void upgradeToCTTAnalysis (
```

```
deadlockPrediction.CodeBlob spec)  throws java.lang.Exception {
if (     accuracy != AnalysisLevel.Null  ||
    spec.accuracy == AnalysisLevel.Null)
    return;
System.out.println ("CT-Interpreting: " +getNodeName()+
      (spec==this ? "." : ("\n        From: " + spec.getNodeName())));
if (spec != this) {
    isPublic  = spec.isPublic;
    isStatic  = spec.isStatic;
    String lt = spec.getLockType();
    if (lt != null)
        setLockType (getContainerType().name() +
                     (lt.endsWith(".class") ? ".class" : ""));
    locals = ((JavaBlob) spec).locals;
    offset = ((JavaBlob) spec).offset;
    assert ((code = ((JavaBlob) spec).code) != null);
    monitorExits = ((JavaBlob) spec).monitorExits;
    assert (! getNodeName().equals (spec.getNodeName()));
    behaviourSpec = spec;
}
cttAnalysisCount++;
for (int i = 0;  i < code.length;  i++)  {
    String line =  code[i];
    if (line.startsWith("invokevirtual\t")   ||    // point 5 above
        line.startsWith("invokespecial\t")   ||    // -- " --
        line.startsWith("invokeinterface\t") ||    // -- " --
        line.startsWith("invokestatic\t")) {       // -- " --
        String ms = line.substring (line.indexOf("; //") + "; //".length());
               ms = line.substring (line.indexOf("Method ") + "Method ".length()
        interpretInvocation (ms, line.startsWith("invokestatic\t"),
                                 line.startsWith("invokespecial\t"),
                                 line.startsWith("invokeinterface\t"),
                                 line.startsWith("invokevirtual\t"), null);
    } else if (line.indexOf("tstatic")==2 &&  //line.indexOf("//Field ")>0) {
                line.substring(line.indexOf("//")+2).trim().startsWith("Field "))
        line = line.substring(line.indexOf("//")+2).trim();
        ⟨ Pick fType and retain attribute-name (96 "Field " ) 84c ⟩
        if (line.indexOf('.') <= 0)
            line = getContainerType().name() + '.' + line;
        /* ClassInfo tv = (ClassInfo) classes.get (fType.name());
        assert (tv != null);
        tv.setAsClassAttribute (line);
        assert ((tv = (ClassInfo) classes.get (getContainerType().name())) != nu
        if (! TypeSignature.basicEncoded.contains (fType.signature())) {
            tv.setClassAttribute (line.substring(line.lastIndexOf('.')+1),
                                  fType.signature());
            // System.out.println ("Re-Setting " +
            //       line.substring(line.lastIndexOf('.')+1) + ":" + fType.signa
        }*/
    }
```

```
        }
        accuracy = AnalysisLevel.CompileTimeType;
    }
    ◇
```

Fragment defined by 86, 91, 94, 98b, 106.
Fragment referenced in 79a.

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 byte-code 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` instruc-

tion. 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.

3. The `interpretInvocation` method implementation operates in two different modes: one mode is operational while jars are being parsed and nodes are thereby being added into the `definedCode` list, and the other mode becomes activated after all input has been exhausted. In this other mode, nodes are not added into the `definedCode` list, but instead into the `varients` list of the relevant node. The second mode is activated by the turning-on of a class-static boolean variable called `exhaustedReadingInputJars`. This is done in the `deadlockPrediction.CodeBlob.main()` method.

5-12-12 In the previous versions of the design / implementation of this analysis, the processing of the *invoke...* family of instructions confused or ill-adviced. Moreover there were three different places in the code that it was being repeated, (and incorrectly!). [**?**] is a well-written article that explains the need for the family of four different varients of the *invoke* byte-code, in Java, and how and when they are used: the distinctions between them, and their specific characteristics. From here-forward our design and the implementation will be corrected as under: There shall be only a single implementation of the (instance method) `interpretInvocation` member of the `CodeBlob` class. It shall take a String argument that is the method-signature as found in the *significant comment* of the input source. Also it will take four boolean arguments, each (respectively) identifying whether the said invocation `isStatic`, `isSpecial`, `isInterface`, or `isVirtual`. The last parameter to this method will be the `TypeStack` instance that is being used to *abstract-interpret* the method body, provided that the interpretation is to be done at the `RunTimeTypeAnalysis`-Level. When the interpretation is to be any less, a null-value is passed in its place, instead.

⟨ *CodeBlob Class Attributes* 98a ⟩ ≡

```
static public boolean
    exhaustedReadingInputJars = false;
```

◇

Fragment defined by 28a, 33a, 35a, 46a, 51, 98a.
Fragment referenced in 6.

⟨ *Method Body Processors* 98b ⟩ ≡

```
static Pattern
u_arith = Pattern.compile ("[ilfd](?:neg)"),           // unary negate
dtc     = Pattern.compile ("[dfil]2[bcsdfil]"),        // Data Type conversion
wideinc = Pattern.compile ("(?:wide)|(?:iinc)"),       // wide / iinc
loads   = Pattern.compile ("[adfil]load(?:_[0123])?"), // Loads
```

```
stores  = Pattern.compile ("[adfil]store(?:_[0123])?"),// Stores
consts  = Pattern.compile ("[dfil]const_m?[012345]"),  // Push Constants
push    = Pattern.compile ("[bs]ipush"),               // byte/short Push
b_arith = Pattern.compile ("[ilfd]((?:add)|(?:sub)|(?:mul)|(?:div)|(?:rem))"),
bitwise = Pattern.compile ("[il]((?:x?or)|(?:and))"),  // bitwise: and, or, xor
shifts  = Pattern.compile ("[il]u?sh[lr]"),            // Shift rght/lft unsighed?
lfdCmp  = Pattern.compile ("[lfd]cmp[lg]?"),           // double-word compares
cmpares = Pattern.compile ("if(?:_[ai]cmp)?((?:eq)|(?:g[et])|(?:l[et])|(?:ne))"),
nullCmp = Pattern.compile ("if(?:non)?null"),          // Null compares
returns = Pattern.compile ("[adfil]?ret(?:urn)?"),     // retrn frm method/subrtn
aloads  = Pattern.compile ("[abcdsilf]aload"),         // Array Load
astores = Pattern.compile ("[abcdsilf]astore"),        // Array Store
gotoJsr = Pattern.compile ("(?:goto)|((?:jsr)(?:_w)?)"),  // goto/jump 2 subrtn
mTarget = Pattern.compile ("((lookup)|(table))(switch)"), // multi-target switch
ldConst = Pattern.compile ("ldc2?(?:_w)?"),            // Load constant ...
stackOp = Pattern.compile ("(?:dup2?(?:_x[12])?)|(?:pop2?)|(?:swap)"),
invokes = Pattern.compile ("(invoke)" +
                           "((?:special)|(?:static)|(?:virtual)|(?:interface))"),
fieldOp = Pattern.compile ("((?:get)|(?:put))((?:field)|(?:static))"),
castOp  = Pattern.compile ("(?:checkcast)|(?:instanceof)"),
newOp   = Pattern.compile ("(?:multi)?a?new(?:array)?"),  // various 'new's.
lockOp  = Pattern.compile ("monitor((?:enter)|(?:exit))"),// Monitor operations
restOp  = Pattern.compile ("a((?:throw)|(?:rraylength)|(?:const_null))"),
c1Match = Pattern.compile ("[BCFISZ]"),                // c1 matcher
c2Match = Pattern.compile ("[DJ]");                    // c2 matcher
static String  c1 = "c1".intern(),  c2 = "c2".intern();

boolean interpret(Stack oStack, int j, Stack tS, Object[] tA,
    List execStartOffsets, List stackAtExecStart, List arrayAtExecStart,
    StringBuffer eCounter) throws Exception {
    Matcher m;
    String bc = (String) code[j];   // System.out.println (j + ":" + 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.pop() == c2);
        } 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)).lookingAt() ||
               (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)).lookingAt()) {
            char tb = bc.charAt(m.start()), tc = bc.charAt(m.end()-1);
            int i = (tc!='d' && tc!='e') ?
```

```
                (tc-'0') : Integer.parseInt(bc.substring(m.end()).trim());
            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
                    tA[i] = tS.pop();
            if (tb == 'a')
                assert (tA[i] != c1  && tA[i] != c2);
            //System.out.println ("pushed / popped: tA[" + i + "] = " + tA[i]);
        }
    } else if ((m = consts.matcher(bc)).lookingAt()  ||
               (m = push.matcher(bc)).lookingAt()) {
        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)).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);
            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)).lookingAt()) {
        tS.pop();
```

```
    if (bc.charAt(m.start()+2) == '_')
        tS.pop();           //below: ctt: control transfer target
    int ctt = Integer.parseInt (bc.substring (m.end()).trim());
    updateBBlist (ctt, tS, tA, execStartOffsets,
                  stackAtExecStart, arrayAtExecStart);
} else if ((m = returns.matcher(bc)).lookingAt())  {
    char tb = bc.charAt(m.start());
    if (tb=='d' || tb=='l')  {
        assert(tS.pop() == c2);          assert(tS.pop() == c2);
    } else if (tb=='a')  {
        String rtRet = (String) tS.pop(); // RTTAnalysis records this
        if (rtReturnTypes != null  ||
            !rtRet.equals (getCTReturnType().signature()))  {
            if (rtRet.equals("Ljava/lang/Object;")  ||
                cExtends (getCTReturnType().name(),
                          new TypeSignature (rtRet, true).name()))
                ;
            else {
                if (rtReturnTypes == null)
                    rtReturnTypes = new TreeSet();
                else
                    System.out.println ("XYZ: Adding runtime: " + rtRet + " for:
                                getReportableName(null));
                rtReturnTypes.add (rtRet);
                System.err.println ("XYZ: Returning " + rtRet +
                                " instead of " + getCTReturnType().signature());
                System.err.println ("XYZ: Method " + getReportableName(null) +
                                " returns " + rtRet);
            }
        }
    } else if (m.group().length() == 7)    assert(tS.pop() == c1);
    else if (m.group().length() == 3) {
        int ctt = Integer.parseInt(bc.substring(m.end()).trim());
            ctt = ((Integer) tA[ctt]).intValue();
        updateBBlist (ctt, tS, tA, execStartOffsets,
                      stackAtExecStart, arrayAtExecStart);
    }
    return false;
} else if ((m = aloads.matcher(bc)).lookingAt()) {
    char tb = bc.charAt(m.start());
    assert (tS.pop() == c1);                    // consume index
    String tmp = (String) tS.pop();             // RTTAnalysis processes this
    assert (tmp.startsWith("[") ||              // confirm array-base
                tmp.equals("Ljava/lang/Object;"));
    if (tb != 'a') {
        if (tb=='d' || tb=='l') {  tS.push(c2);    tS.push(c2);  }
        else                       tS.push(c1);
    } else
        tS.push(tmp.startsWith("[") ? tmp.substring(1) : tmp);
        // consume array-base & re-instate element-type
```

```
    //System.out.println ("Popped: " + tmp + "; Pushed: " + tS.peek());
} else if ((m = astores.matcher(bc)).lookingAt())  {
    char tb = bc.charAt(m.start());
    if (tb == 'a')              tS.pop(); // RTTAnalysis processies this
    else if (tb=='d' || tb=='l') {
        assert(tS.pop() == c2);  assert(tS.pop() == c2);
    } else                       assert(tS.pop() == c1);
    assert(tS.pop() == c1);
    String tmp = (String) tS.pop();
    assert (tmp.startsWith("[") || tmp.equals("Ljava/lang/Object;"));
} else if ((m = gotoJsr.matcher(bc)).lookingAt())  {
    if (bc.charAt(m.start()) == 'j') {
        assert (j < offset.length-1);
        tS.push(offset[j+1]);
    }
    int ctt = Integer.parseInt (bc.substring (m.end()).trim());
    updateBBlist (ctt, tS, tA, execStartOffsets,
                  stackAtExecStart, arrayAtExecStart);
    return false;
} else if ((m = mTarget.matcher(bc)).lookingAt()) {
    while (true) {
        int cln = bc.indexOf(':'),  sCln = bc.indexOf(';');
        if (sCln<0) sCln = bc.indexOf(" }");
        if (cln>0 && sCln>0 && cln<sCln) {
            int ctt = Integer.parseInt (bc.substring (cln+1, sCln).trim());
            updateBBlist (ctt, tS, tA, execStartOffsets,
                          stackAtExecStart, arrayAtExecStart);
            bc = bc.substring(sCln+1);
        } else
            break;
    }
}  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());
        String tmp = bc.substring(bc.indexOf("//")+2).trim();
        int cInd = tmp.indexOf ("class "), sInd = tmp.indexOf ("String");
        tS.push(sInd>=0 ? "Ljava/lang/String;" :
                (cInd<0 ? c1 : "Ljava/lang/Class;"  //));
                 +":L"+tmp.substring(cInd + "class ".length()).trim()+".class;")
    }
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) {  //dup
            assert ((ts = tS.peek()) != c2);  tS.push(ts);
        } else if (m.group().length() == 4) {    //dup2
            if ((ts=tS.pop()) == c2) {
```

```
                    assert (tS.peek() == c2);  tS.push(c2);
                    tS.push(c2);               tS.push(c2);
                } else {
                    assert((tsn=tS.peek()) !=  c2);  tS.push(ts);
                    tS.push(tsn);                    tS.push(ts);
                }
            } else if (m.group().length()==6 && bc.charAt(m.end()-1)=='1') {
                assert ((ts=tS.pop()) != c2);        //dup_x1
                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') {
                assert ((ts=tS.pop()) != c2);        //dup_x2
                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') {
                ts = tS.pop();       tsn = tS.pop(); //dup2_x1
                assert((tsnn=tS.pop()) != c2);
                tS.push(tsn);        tS.push(ts);    tS.push(tsnn);
                tS.push(tsn);        tS.push(ts);
            } else if (m.group().length()==7 && bc.charAt(m.end()-1)=='2') {
                ts = tS.pop();       tsn = tS.pop(); //dup2_x2
                tsnn = tS.pop();     tsnnn = tS.pop();
                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 "swap".
            assert ((ts = tS.pop()) != c2);   assert ((tsn = tS.pop()) != c2);
            tS.push(tsn);                     tS.push(ts);
        }
    } else if ((m = invokes.matcher(bc)).lookingAt()) {
        boolean isInterface = m.group().endsWith ("interface");
        String sstr = isInterface ? "InterfaceMethod " : "Method ";
        String ms = bc.substring(bc.indexOf("//") + 2).trim();
               ms = ms.substring(ms.indexOf(sstr) + sstr.length());
               ms = ms.substring (0, ms.indexOf(':')).replace ('/', '.') +
                    ms.substring (ms.indexOf(':'));
        JavaBlob tos = (JavaBlob)oStack.peek();
        deadlockPrediction.CodeBlob tB  = // Target Behaviour invoked below !!
        tos.interpretInvocation (ms, m.group().endsWith ("static"),
```

```
                                    m.group().endsWith ("special"), isInterface,
                                    m.group().endsWith ("virtual"), tS);
        if (tB != null  &&  tB.rtReturnTypes != null  &&
           !tB.getCTReturnType().signature().startsWith("V"))  {
            String ctrt = (String) tS.pop();
            for (Iterator ii = tB.rtReturnTypes.iterator();  ii.hasNext();  ) {
                tS.push (ii.next());
                updateBBlist (offset[j+1], tS, tA, execStartOffsets,
                          stackAtExecStart, arrayAtExecStart);
                tS.pop();
            }
            tS.push (ctrt);
            return false;
        }
    } else if ((m = fieldOp.matcher(bc)).lookingAt())  {
        Object ts = null, tss = null;
        ClassInfo objT = null;
        String attT = null, attN = bc.substring(bc.lastIndexOf(' ')+1, bc.indexOf(':
        String  ctT = bc.substring(bc.indexOf(':')+1).trim(), cn;
        if (attN.indexOf('.') >= 0)
            attN = attN.substring (attN.lastIndexOf('.')+1);
        if (bc.charAt(m.start()) == 'g') {
            if (bc.charAt(m.start()+3) == 'f') {
                // System.out.println ("getfield container fetched: " + ((String) tS
                assert ((ts = tS.pop()) != c1  &&  ts != c2);
                assert (((String)ts).startsWith("L"));
                cn = ((String)ts).substring (1, ((String)ts).length()-1).replace('/'
            } else
                cn = getContainerType().name();
            if (ensureField (cn, attN, bc.charAt(m.start()+3) == 's', ctT))  {
                assert ((objT = (ClassInfo) classes.get (cn)) != null);
                attT = (String) ((bc.charAt(m.start()+3) == 'f')  ?
                      objT.instanceAttributes.get(attN) : objT.classAttributes.get(a
            }
            tS.push (abstractTenc (moreConfiningOf (attT, ctT)));
            // System.out.println ("getField/static Fetched: " + tS.peek());
            if (tS.peek().equals(c2))   tS.push(c2);
        } else {
            if ((ts=tS.pop()) == c1)
                assert (c1Match.matcher(ctT).lookingAt());
            if (ts==c2) {            assert (tS.pop()==c2);
                assert (c2Match.matcher(ctT).lookingAt());
            }
            if (bc.charAt(m.start()+3) == 'f')
                assert ((tss = tS.pop())!=c1   &&   tss!=c2);
            // if (!((String)ts).endsWith(c1)  &&  !((String)ts).endsWith(c2)) {
            if (ts != c1  &&  ts != c2) {
                cn = (bc.charAt(m.start()+3) == 's') ? getContainerType().name()
                   : ((String)tss).substring (1, ((String)tss).length()-1).replace (
                if (ensureField (cn, attN, bc.charAt(m.start()+3) == 's', ctT))  {
```

```
                assert ((objT = (ClassInfo) classes.get (cn)) != null);
                if (bc.charAt(m.start()+3) == 'f')
                    objT.instanceAttributes.put (attN, ts);
                else
                    objT.classAttributes.put (attN, ts);
            }
        }
    }
} 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("//") + 2).trim();
               t =  t.substring( t.indexOf("class ") + "class ".length()).trim()
        tS.push (t.charAt(0) == '"' ? t.substring(1, t.length()-1) : ("L" + t + '
    }
} else if ((m = newOp.matcher(bc)).lookingAt())  {
    String typePreFix = "", tmp = null;
    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 += "[";
    }
    if (m.group().equals("newarray"))
        tmp = (new TypeSignature(bc.substring(m.end()).trim(), false)).signature
    else {
        tmp = bc.substring(bc.indexOf("//") + 2).trim();
        tmp = tmp.substring(tmp.indexOf("class ") + "class ".length()).trim();
    }
    //String tmp = (m.group().equals("newarray"))
    //    ? (new TypeSignature(bc.substring(m.end()).trim(), false)).signature()
    //    : bc.substring(bc.indexOf("//class ") + "//class ".length()).trim();
    tS.push (typePreFix + (m.group().equals("newarray") ? abstractTenc (tmp)
        : (tmp.charAt(0)=='"' ? tmp.substring(1, tmp.length()-1) : ('L' + tmp +'
} else if ((m = lockOp.matcher(bc)).lookingAt())  {
    JavaBlob po = (JavaBlob) oStack.peek();
    if (m.group().endsWith("enter")) {
        String lock = (String)tS.pop();
        if (lock.indexOf(":") > 0)
            lock = lock.substring (lock.indexOf(":")+1);
        assert (lock.startsWith("L") || lock.startsWith("["));
        String key = (po.getNodeName() +'#'+ lock +'#'+ eCounter.charAt(0)).inte
        oStack.push (new JavaBlob(po, key, (lock.endsWith(c1) || lock.endsWith(c
```

```
                                                    lock : new TypeSignature (lock, true)
                eCounter.setCharAt(0, (char)(eCounter.charAt(0) + 1));
            } else {
                //System.out.println ("monitorExits: " + monitorExits +
                //    ";\nCompare: " + (String)tS.peek() + ":" + po.getLockType() +
                //    ";\nnextOffset: " + offset[j+1]);
                if (monitorExits.contains(offset[j+1])) {
                    String lock = (String) tS.peek();
                    if (lock.indexOf(":") > 0)
                        lock = lock.substring (lock.indexOf(":")+1);
                    assert (lock.startsWith("L") || lock.startsWith("["));
                    assert (po.getLockType().equals (new TypeSignature (lock, true).name
                            lock.equals (po.getLockType()) ||
                            lock.equals (new TypeSignature (po.getLockType(), false).sig
                    oStack.pop();
                }
                tS.pop();
            }
        } else if ((m = restOp.matcher(bc)).lookingAt()) {
            if (! m.group().equals("aconst_null")) {
                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);
        return true;
    }
    ◇
```

Fragment defined by 86, 91, 94, 98b, 106.
Fragment referenced in 79a.

⟨ *Method Body Processors* 106 ⟩ ≡

```
    public Object[] getTypeArrayFromParams (int locals)  {
        // System.out.println ("Locals here: " + locals + "isStatic: " + isStatic);
        java.lang.Object[] retval = new java.lang.Object[locals];
        TypeSignature[] pList = getParams();
        if (((pList==null ? 0 : pList.length) + (isStatic ? 0 : 1)) > locals)
            System.out.println ("Insufficient locals specified: " + getNodeName());
        if (! isStatic)
            retval[0] = getContainerType().signature();
```

```
    for (int k = 0, j = isStatic ? 0 : 1;  k < getParamCount();  k++, j++) {
        retval[j] = abstractTenc (pList[k].signature());
        if (retval[j] == c2) retval[++j] = c2;
    }
    // System.out.println ("initialisation fetches:");
    // for (int kk = 0;  kk<locals && retval[kk]!=null; kk++)
    //     System.out.println (retval[kk].toString());
    return retval;
}

public static void printArrayNdStack (Object[] a, Stack s) {
    for (int i = 0;  i < a.length;  i++)
        System.out.println ("a["+i+"] = " + (a[i]!=null ? a[i].toString() : "null"))
    Stack ss = new Stack();
    while (!s.empty()) {
        System.out.println ("s = " + (s.peek()!= null ? s.peek() : null));
        ss.push (s.pop());
    }
    while (!ss.empty())
        s.push (ss.pop());
}

public static String abstractTenc (String type)  {
    if (type == null  ||  type.equals(c1)  ||  type.equals(c2)  ||
        (type.startsWith("L") && type.endsWith(";")))
        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 {
        System.out.println ("Bad input: " + type + " ?");
        assert (false);
    }
    return null;
}
public static String moreConfiningOf (String rt, String ct) {
    String retval;
    if (rt == null  ||  (rt.length()==2  &&  ct.length()==1))
        retval = ct;
    else if (rt.startsWith("[")  ||  ct.startsWith("[")) {
        int rti = rt.lastIndexOf("["), cti = ct.lastIndexOf("[");
        retval = (rti > cti  ?  rt
                             : (rti < cti  ?  ct
                                           :  (rt.substring (0, rti+1) +
             moreConfiningOf (rt.substring(rti+1), ct.substring(cti+1)))));
    } else {
        //int rti = rt.length(), cti = ct.length();
```

```
        //retval = (rti >= cti) ?  rt  :  ct));
        retval = betterDefined (new TypeSignature(rt, true),
                                new TypeSignature(ct, true)).signature();
    }
    // System.out.println ("rt: " + rt + "; ct: " + ct + "; retval: " + retval);
    return retval;
};

public static boolean cExtends (String c1, String c2) {
    for (ClassInfo ci = null;  c1 != null;
         c1 = ((ci = (ClassInfo)classes.get(c1)) == null) ? null : ci.parentType)
         if (c1.equals(c2))
             return true;
         // else
         //     System.out.println (c1 + " != " + c2);
    return false;
};

public static boolean iExtends (String i1, String i2) {
    Stack il = new Stack(); // list of interfaces represented/extended by i1
    String[] ie = null;     // temp: set of interfaces extended by an interface
    for (il.push (i1);  !il.empty();  )  {
        String iN = (String) il.pop(); // interface Name
        if ((ie = (String[]) interfaces.get(iN)) != null)
            for (int j = 0;  j < ie.length;  j++)
                il.push (ie[j]);
        //System.out.println (iN + " ?= " + i2);
        if (iN.equals(i2))
            return true;
    }
    return false;
};

public static String rmSqBracs (String cn) {
    int ocsb = -1;
    System.out.println ("XXX Removing Square brace-sets: " + cn);
    while ((ocsb = cn.indexOf("[]")) >= 0)
        cn = cn.substring(0, ocsb) + cn.substring(ocsb+2);
    return cn;
};
public static boolean cImplementsI (String c, String i) {
    Stack iS = new Stack ();
    for (ClassInfo ci = (ClassInfo) classes.get(c);  c != null;
         c = ((ci = (ClassInfo)classes.get(c)) == null) ? null : ci.parentType)
         if (ci.interfacesImplemented != null)
             for (int j = 0;  j < ci.interfacesImplemented.length;  j++)
                 iS.push (ci.interfacesImplemented[j]);
    for (String[] ie = null;  !iS.isEmpty();  )  {
        String ii = (String) iS.pop();
        if (ii.equals(i))    return true;
```

```
        if ((ie = (String[]) interfaces.get(ii)) != null)
            for (int j = 0;  j < ie.length;  j++)
                iS.push (ie[j]);
    }
    while (!iS.isEmpty())
        if (i.equals((String)iS.pop()))
            return true;
        // else System.out.println (i + " != " + i2);
    return false;
};

public static TypeSignature betterDefined (TypeSignature t1, TypeSignature t2)  {
    assert (t1 != null  &&  t2 != null);
    String cn1 = t1.name(), cn2 = t2.name();
    if (cn1.indexOf("[]") >= 0)  cn1 = rmSqBracs (cn1);
    if (cn2.indexOf("[]") >= 0)  cn2 = rmSqBracs (cn2);
    Collection cL = classes.keySet(), iL = interfaces.keySet();
    //System.out.println ("Comparing Types: " + cn1 + "::" + cn2);
    if (cn1.startsWith ("java.lang.Class;")  ||
        cn2.startsWith ("java.lang.Class;"))
        return new TypeSignature ("Ljava/lang/Class;", true);
    else if (cL.contains (cn1)  &&  cL.contains (cn2)) {
        //System.out.println ("betterDefined:case 1!!");
             if (cExtends (cn1, cn2))  return t1;
        else if (cExtends (cn2, cn1))  return t2;
        else return t2; // works well with moreConfiningOf!!
    } else if (iL.contains(cn1)  &&  iL.contains(cn2)) {
        // System.out.println ("bD2: "+ cn1 + "---" + cn2);
             if (iExtends (cn1, cn2))  return t1;
        else if (iExtends (cn2, cn1))  return t2;
        else                           return t2;
    } else if ((cL.contains (cn1) && iL.contains (cn2))  ||
               (iL.contains (cn1) && cL.contains (cn2)))  {
        //System.out.println ("betterDefined:case 3!!");
        String cN = cL.contains(cn1) ? cn1 : cn2;
        String iN = iL.contains(cn1) ? cn1 : cn2;
        if (cImplementsI (cN, iN))
             return cN.equals(cn1) ? t1 : t2;
        else if (cN.equals ("java.lang.Object"))
             return iN.equals(cn1) ? t1 : t2;
        else return t2; // assert (false);
    } else if (cn1.equals ("c1")  ||  cn1.equals ("c2"))
        return t2;
    return t2;  // t1;
}
public static boolean isArelation (TypeSignature tOfo, String t)  {
    assert (tOfo != null  &&  t != null);
    String cn = tOfo.name();
    Collection cL = classes.keySet(), iL = interfaces.keySet();
         if (cn.equals (t))
```

```
        return true;
    else if (cL.contains(cn) && cL.contains(t) && cExtends (cn, t))
        return true;
    else if (iL.contains(cn) && iL.contains(t) && iExtends (cn, t))
        return true;
    else if (cL.contains(cn) && iL.contains(t) && cImplementsI (cn, t))
        return true;
    else if (iL.contains(cn) && cL.contains(t) && cImplementsI (t, cn))
        return true;
    if (t.equals("java.lang.Object"))  return true;
    //System.out.println (cn + " is not of type " + t);
    return false;
}

public deadlockPrediction.CodeBlob
       interpretInvocation (String tMethod, boolean isStatic, boolean isSpecial,
                            boolean isInterface, boolean isVirtual, Stack typeStack)
    if (tMethod.startsWith("\"")  &&  tMethod.indexOf("\".clone:")>0) {
        if (typeStack != null) {
            String tos = (String) typeStack.peek();
            assert (tos != c1  &&  tos != c2);
            typeStack.push ("Ljava/lang/Object;");
        }
        return null;
    }
    TypeSignature definingClass = (behaviourSpec == null) ?
             getContainerType() : behaviourSpec.getContainerType();
    assert (MethodSignature.isFullySpecified (tMethod));
    MethodSignature tM = new MethodSignature (tMethod);
    String rt  = tM.retType().signature();
    String cms = tM.signature();
    JavaBlob rb = null;
    if (typeStack == null) { // conducting CompileTimeType-Level interpretation
        if (! notDefinedCode.contains (cms)) {
            if (isInterface || tobeDefinedCode.get(cms)!=null)
                noteCallTo (cms);
            else {
                JavaBlob cb = (JavaBlob) definedCode.get (cms);
                if (cb == null)
                    cb = new JavaBlob (cms);
                if (cb != null)
                    noteCallTo (cb);
            }
        }
    }  else  { // conducting RunTimeType-Level interpretation
        String rms = "", ccsNm = null, rcms = null;
        for (int i = tM.paramCount();  --i >= 0;  )  {
            String pt = tM.paramList()[i].signature();
            if (c1Match.matcher(pt).lookingAt()  ||  pt.startsWith ("[")) {
                typeStack.pop();                            rms = pt + rms;
```

```
    } else if (c2Match.matcher(pt).lookingAt()) {
        typeStack.pop();      typeStack.pop();      rms = pt + rms;
    } else if (pt.equals("Ljava/lang/Class;")) {
        TypeSignature actual = new TypeSignature ((String) typeStack.pop(),
        if (! actual.name().startsWith("java.lang.Class")  &&
            ! actual.name().equals("java.lang.Object"))
            System.out.println ("InterpretInvocation: " + actual.name() + "#
                                " When " + getNodeName() + " Invoking " + tM
        rms  = "Ljava/lang/Class;" + rms;
    } else if (pt.startsWith("L")  && pt.endsWith(";")) {
        TypeSignature actual = new TypeSignature ((String) typeStack.pop(),
        if (!isArelation (actual, tM.paramList()[i].name())  &&
            !isArelation (tM.paramList()[i], actual.name()))
            System.out.println ("InterpretInvocation: " + actual.name() + "/
                                " When " + getNodeName() + " Invoking " + tM
        rms  = (pt.endsWith(";") ?
                betterDefined (actual, tM.paramList()[i]).signature() : pt) +
    } else
        assert (false);
}
rms = "." + tM.methodName() + ":(" + rms + ")" + rt;
if (!isStatic)  {
    TypeSignature tOfo = new TypeSignature ((String)typeStack.peek(), true);
    if (tOfo.signature().startsWith("Ljava/lang/Class;")) {
        if (!tM.className().name().equals("java.lang.Class")  &&
            !tM.className().name().equals("java.lang.Object"))
            System.out.println ("InterpretInvocation1: " + tOfo.name() +"#-#
                                 tM.className().name() + " When " + getNodeNa
                                " Invoking " + tM.signature());
    } else if (!isArelation (tOfo, tM.className().name())  &&
        !isArelation (tM.className(), tOfo.name()))
        System.out.println ("InterpretInvocation2: " + tOfo.name() +"#-#"+
                             tM.className().name() + " When " + getNodeName(
                            " Invoking " + tM.signature());
}
rms = (isStatic ? tM.className().name()
       : (((ccsNm = (String) typeStack.pop()).endsWith(c1) || ccsNm.endsWith
          ? ccsNm : betterDefined (new TypeSignature (ccsNm, true),
                                   tM.className()).name())) + rms;
if (isVirtual || isInterface) {
    rcms  = rms.substring  (0, rms.indexOf(":"));
    rcms  = rcms.substring (0, rcms.lastIndexOf(".")) + ".";
    rcms += (tM.methodName() + ":" + tM.prtEncoding());
}
if (!(notDefinedCode.contains(cms) && (isSpecial || isStatic)))  {
    if ( rms.indexOf("[]") >= 0)     rms = rmSqBracs ( rms);
    if (isSpecial || isStatic) {
        JavaBlob cb = (JavaBlob) definedCode.get (cms);
        if (cb != null) {
            rb = (JavaBlob) getCodeOrVarientBlob(cb.getNodeName(), rms);
```

```
                    if (rb == null)
                        rb = new JavaBlob (cb, rms);
                    noteCallTo (rb);
                } else
                    System.out.println ("XXX Cant find non-native static/special cms
            } else {
                if (rcms.indexOf("[]") >= 0)    rcms = rmSqBracs (rcms);
                JavaBlob cb = (JavaBlob) definedCode.get (rcms);
                if (isInterface  &&  tobeDefinedCode.get(cms) == null)
                    System.out.println ("XXX How come " + cms + " is not in tobeDefi
                if (cb == null)
                    cb = (JavaBlob) getBspecForIVcall (new MethodSignature (rcms));
                if (cb != null) {
                    rb = (JavaBlob) getCodeOrVarientBlob  (cb.getNodeName(), rms);
                    if (rb == null)
                        rb = new JavaBlob (cb, rms);
                    if (isInterface)  noteCallTo (cms, rb);
                    else              noteCallTo (cb,  rb);
                }
            }
        }
        if (rt.charAt(0) != 'V') {
            typeStack.push (abstractTenc (rt));
            if (((String)typeStack.peek()).equals(c2))    typeStack.push(c2);
        }
    }
    return rb;
}

public void  updateBBlist (int ctt, Stack tS, Object[] tA,
       List execStartOffsets, List stackAtExecStart, List arrayAtExecStart)  {
    if (indexOfOffset(ctt) != -1) {
        int i = 0;
        while (i < execStartOffsets.size()) {
             int temp = ((Integer)execStartOffsets.get(i)).intValue();
                  if (ctt >  temp)    i++;
             else if (ctt == temp)    return;
             else                     break;
        }
        execStartOffsets.add(i, ctt);
        stackAtExecStart.add(i, tS.clone());
        arrayAtExecStart.add(i, tA.clone());
    }
}

private int indexOfOffset (int ctt) {
    int start = 0, end = offset.length - 1, midPt;
    while (start <= end) {
        midPt = (start + end) / 2;
        if (offset[midPt] == ctt) {
```

```
                return midPt;
            } else if (offset[midPt] < ctt) {
                start = midPt + 1;
            } else {
                end = midPt - 1;
            }
        }
        return -1;
    };
    ◇
```

Fragment defined by 86, 91, 94, 98b, 106.
Fragment referenced in 79a.

1.9 MappedMXBeanType API deadlocks predicted by the analysis

The analysis of the Jdk-version-6 to detect only 2- and 3-cycles reports 144 cycles, of which 15 2-cycles, and 20 3-cycles are in the `MappedMXBeanType` API. A sample 2-cycle is reproduced below. All the (15 + 20 = 35) cycles are on locks aquired by class-static methods. For brevity, the repetitive prefix ”sun.management.” has been dropped, wherever `MappedMXBeanType` appears below.

```
<Cycle-2 locks="MappedMXBeanType$ListMXBeanType.class
                MappedMXBeanType$MapMXBeanType.class">
  <Thread-1>
    MappedMXBeanType$ListMXBeanType.getMappedType:
        (Ljava/lang/reflect/Type;)LMappedMXBeanType;
    MappedMXBeanType.newMappedType:
        (Ljava/lang/reflect/Type;)LMappedMXBeanType;
    MappedMXBeanType$MapMXBeanType."init":
        (Ljava/lang/reflect/ParameterizedType;)V
    MappedMXBeanType$MapMXBeanType.getMappedType:
        (Ljava/lang/reflect/Type;)LMappedMXBeanType;
  </Thread-1>
  <Thread-2>
    MappedMXBeanType$MapMXBeanType.getMappedType:
        (Ljava/lang/reflect/Type;)LMappedMXBeanType;
    MappedMXBeanType.newMappedType:
        (Ljava/lang/reflect/Type;)LMappedMXBeanType;
    MappedMXBeanType$ListMXBeanType."init":
        (Ljava/lang/reflect/ParameterizedType;)V
    MappedMXBeanType$ListMXBeanType.getMappedType:
        (Ljava/lang/reflect/Type;)LMappedMXBeanType;
```

```
  </Thread-2>
</Cycle-2>
```

The pairs of 2-Cycle locks involved in *pottential* deadlocks are listed below. The details of the thread-stacks as in the prediction above are ommited below for brevity.

```
MappedMXBeanType$ListMXBeanType  MappedMXBeanType$MapMXBeanType
MappedMXBeanType$ListMXBeanType  MappedMXBeanType$ArrayMXBeanType
MappedMXBeanType$ListMXBeanType  MappedMXBeanType$GenericArrayMXBeanType
MappedMXBeanType$ListMXBeanType  MappedMXBeanType
MappedMXBeanType$ListMXBeanType  MappedMXBeanType$CompositeDataMXBeanType
MappedMXBeanType$MapMXBeanType   MappedMXBeanType$ArrayMXBeanType
MappedMXBeanType$MapMXBeanType   MappedMXBeanType$GenericArrayMXBeanType
MappedMXBeanType$MapMXBeanType   MappedMXBeanType
MappedMXBeanType$MapMXBeanType   MappedMXBeanType$CompositeDataMXBeanType
MappedMXBeanType$ArrayMXBeanType MappedMXBeanType$GenericArrayMXBeanType
MappedMXBeanType$ArrayMXBeanType MappedMXBeanType
MappedMXBeanType$ArrayMXBeanType MappedMXBeanType$CompositeDataMXBeanType
MappedMXBeanType$GenericArrayMXBeanType MappedMXBeanType
MappedMXBeanType$GenericArrayMXBeanType MappedMXBeanType$CompositeDataMXBeanType
MappedMXBeanType  MappedMXBeanType$CompositeDataMXBeanType
```

In the rest of this section, our purpose is to synthesize a Java program that can deadlock its executing JVM in the various manners as predicted by our analysis, above. We begin with first developing an understanding of the intended purpose and the typical usage of the said API.

1.9.1 The MappedMXBeanType API

The primary purpose of this API is to firstly create a `MappedMXBeanType` object corresponding to a given `JavaType`. Having done this, it then becomes possible to render any data-object of the `JavaType` into its corresponding `OpenData`, and vise-versa. In this sense, the capability reminds one of the ASN.1/C++ Specific, and Generic Interfaces. The Necessary condition is that all the attributes of the `JavaType` must belong within a subset of the Java-Typing expressive capabilities.

Specifically, the `MappedMXBeanType` type is an abstract base-class that is sub-classed from by classes like `MappedMXBeanType$MapMXBeanType`. Its various subclasses are concrete. This API is intended for use by code that wishes to, for instance, display the state of objects hosted in the JVM's MBean container, in the UI of the JConsole-application. A typical use-case for this API is the GC-info that is displayed in the JConsole, for which the data is sourced from the `MemoryUsageCompositeData` object hosted in the JVM's MBean-container, whose state is in-turn maintained consistant with the relevent sub-set state of the GC subsystem of the JVM, itself.

On closer examination of the File:sun/management/MappedMXBeanType.java it appears that, for instance, creating the `MappedMXBeanType` object corresponding to some List of Maps along one thread-of-control, and another corresponding to some

other Map < Name-represented-as-a-String, Value-represented-by-a-List >, along another thread-of-control can deadlock the executing JVM, as prediced in the cycle detailed in the begining of this section.

The page: http://download.oracle.com/javase/tutorial/jmx/mbeans/mxbeans.html illustrates sample code that deploys an MXBean in the JVM-bean-container. The example comprises of an interface definition, and a Class definition, that implements the interface. An instance-object of the Class, deployed in the JVM-bean-container responds to method-invocation requests sent from a JConsole application connected to the hosting-JVM.

Using the observation regarding the cycle involving Lists, and Maps, and proceeding along the lines of the sample code, the following interface definition could be used to begin with:

"ListMapMXBean.java" 115a≡

```
package deadlock;
public interface ListMapMXBean {
    public void goDeadlock();
}
◇
```

"ListMap.java" 115b≡

```
package deadlock;
import java.lang.*;
import java.lang.reflect.*;
import java.util.*;

public class ListMap implements ListMapMXBean {
  public static Map  <String, List<Integer>> odd1to10;
  public static List <Mapper> indexToOdd;

  public ListMap () {
    int[] odds = {1, 3, 5, 7, 9};
    odd1to10 = new TreeMap <String, List<Integer>> ();
    List <Integer> oInts = new LinkedList <Integer> ();
    for (int i : odds)
      oInts.add (new Integer (i));
    odd1to10.put ("Odds1to10", oInts);

    indexToOdd = new LinkedList < Mapper > ();
    Map <Integer, Integer> oddMap = new TreeMap <Integer, Integer> ();
    for (int i = 0;  i < odds.length;  i++)
       oddMap.put (new Integer(i+1), new Integer (odds[i]));
    indexToOdd.add (new Mapper (oddMap));
  }

  public void goDeadlock() {
```

```
        new Thread() {
          public void run() {
            try {
              System.out.println ("1: the containing class: " +
                  Class.forName("deadlock.ListMap").getField("odd1to10").getGenericType(
              sun.management.MappedMXBeanType.toOpenType (
                Class.forName("deadlock.ListMap").getField("odd1to10").getGenericType())
            } catch (java.lang.Exception e) {
              System.out.println("1 Exception : " + e);
            }
            System.out.println("1 Exiting successfully !");
          };
        }.start();
        new Thread() {
          public void run() {
            try {
              System.out.println ("2: the containing class: " +
                  Class.forName("deadlock.ListMap").getField("indexToOdd").getGenericType
              //sun.management.MappedMXBeanType.toOpenTypeData (
              //  ListMap.indexToOdd, Class.forName("deadlock.ListMap").getField("indexTo
                sun.management.MappedMXBeanType.toOpenType (
                  Class.forName("deadlock.ListMap").getField("indexToOdd").getGenericType
            } catch (java.lang.Exception e) {
              System.out.println("2 Exception : " + e);
              e.printStackTrace();
            }
            System.out.println("2 Exiting successfully !");
          };
        }.start();
      }
    }
    ◇
```

"Mapper.java" 116≡

```
    package deadlock;
    import java.util.*;
    import javax.management.openmbean.*;
    import java.beans.ConstructorProperties;

    public class Mapper implements java.io.Serializable {
        private Map <Integer, Integer> odds;

        @ConstructorProperties({"odds"})
        public Mapper (Map <Integer, Integer> odds) {
            this.odds = odds;
        }
```

```
        public Map <Integer, Integer> getOdds() {
            return odds;
        }
    }
    ◇
```

"QueueSample.java" 117a≡

```
    package deadlock;
    import java.beans.ConstructorProperties;
    import java.util.Date;

    public class QueueSample {
        private final Date date;
        private final int size;
        private final String head;

        @ConstructorProperties({"date", "size", "head"})
        public QueueSample(Date date, int size, String head) {
            this.date = date;
            this.size = size;
            this.head = head;
        }

        public Date getDate()   { return date;  }
        public int getSize()    { return size;  }
        public String getHead() { return head;  }
    }
    ◇
```

"Main.java" 117b≡

```
    package deadlock;

    import java.lang.management.ManagementFactory;
    import javax.management.MBeanServer;
    import javax.management.ObjectName;

    public class Main {
      public static void main(String[] args) throws Exception {
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        mbs.registerMBean(new ListMap(), new ObjectName("deadlock:type=ListMap"));
        System.out.println("Waiting...");
        Thread.sleep(Long.MAX_VALUE);
      }
    }
    ◇
```

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.

1.10 com.sun.media.sound API deadlocks predicted by the analysis

Of the 144 predictions of 2-, and 3- cycles, 5 happen to be in a single library: com.sun.media.sound. In the reproduction below, we have replaced the prefix "com.sun.media.sound" with ".." for brevity.

```
<Cycle-2 locks="..RealTimeSequencer ..RealTimeSequencer$PlayThread">
<Cycle-2 locks="..RealTimeSequencer ..MidiUtils$TempoCache">
<Cycle-2 locks="..MidiUtils$TempoCache ..RealTimeSequencer$DataPump">
<Cycle-3 locks="..RealTimeSequencer ..RealTimeSequencer$PlayThread
                ..MidiUtils$TempoCache">
<Cycle-3 locks="..RealTimeSequencer ..RealTimeSequencer$DataPump
                ..MidiUtils$TempoCache">
```

1.11 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.

1.12 Fragment indexes

⟨ A Short Java Program 77a ⟩ Not referenced.
⟨ Algorithmic components to transform the graphs, and analyse 31a, 34ab, 46c, 48, 53b ⟩ Referenced in 6.
⟨ Assert nodeName is not used-up 33b ⟩ Referenced in 29a.
⟨ Class definition Processor 80 ⟩ Referenced in 79a.
⟨ ClassInfo Constructors 63a ⟩ Referenced in 8b.
⟨ ClassInfo Instance Attributes 52a, 61a ⟩ Referenced in 8b.
⟨ ClassInfo Instance Methods 63b ⟩ Referenced in 8b.
⟨ CodeBlob Class Attributes 28a, 33a, 35a, 46a, 51, 98a ⟩ Referenced in 6.
⟨ CodeBlob Constructors 29a, 30ab, 32a, 37 ⟩ Referenced in 6.
⟨ CodeBlob Instance Attributes 26, 27b, 28bc, 36a, 46b, 52b ⟩ Referenced in 6.
⟨ CodeBlob Instance Methods 27a, 32b, 36b, 40, 41, 47, 50, 55 ⟩ Referenced in 6.
⟨ CodeBlob Instance unnecessary Methods 31b ⟩ Not referenced.
⟨ Field Name & Type Processor 84a ⟩ Referenced in 79a.
⟨ Globally accessable state for sharing data amongst threads 72a ⟩ Referenced in 69.
⟨ Interface declaration Processor 79b ⟩ Referenced in 79a.
⟨ JavaBlob Constructors 71 ⟩ Referenced in 69.
⟨ JavaBlob Instance Attributes 85 ⟩ Referenced in 69.
⟨ JavaFile Opener 8c ⟩ Referenced in 6, 7ab, 8ab.
⟨ Main loop to parse text-rendering 78 ⟩ Referenced in 79a.
⟨ Main method to execute algorithms as per the steps, to print cycles, if any 70 ⟩ Referenced in 69.
⟨ Method Body Processors 86, 91, 94, 98b, 106 ⟩ Referenced in 79a.
⟨ Method Header Processor 82 ⟩ Referenced in 79a.
⟨ Methods to parse text-rendering and create initial call-graph structures 79a ⟩ Referenced in 69.
⟨ Methods to process invocation & invoke javap to create text-rendering 72b, 73, 75, 76 ⟩ Referenced in 69.
⟨ MethodSignature component getters 23 ⟩ Referenced in 7a.
⟨ MethodSignature Constructors 21 ⟩ Referenced in 7a.
⟨ OTlock Instance Attributes 38b ⟩ Referenced in 8a.
⟨ Pick fType and retain attribute-name 84c ⟩ Referenced in 84a, 86, 94.
⟨ Print list of Attributes 61e ⟩ Referenced in 61b.
⟨ Print list of Attributes associated with instance-locks 61b ⟩ Not referenced.
⟨ Report Statistics 53a ⟩ Referenced in 70.
⟨ Store cycle-signature and check to avoid repeats 60 ⟩ Referenced in 62.
⟨ Take note of synchronized use 35b ⟩ Referenced in 32ab.
⟨ The text-rendering of MLtest.class 77b ⟩ Not referenced.
⟨ The Type lock order graph 38a ⟩ Referenced in 8a.
⟨ TLOG Cycle-detection and reporting 57, 58, 62 ⟩ Referenced in 8a.
⟨ TypeSignature Constructors 24a ⟩ Referenced in 7b.
⟨ TypeSignature Methods 25 ⟩ Referenced in 7b.
⟨ TypeSignature Static Initialisation 24b ⟩ Referenced in 7b.

1.13 File indexes

`"B.java"` Defined by 12.
`"ClassInfo.java"` Defined by 8b.

1.14 User-specified indexes