8

JaDA – the Java Deadlock Analyzer

Abel Garcia and Cosimo Laneve

Department of Computer Science and Engineering, University of Bologna – INRIA FOCUS, Mura Anteo Zamboni 7, 40127, Bologna, Italy

Abstract

JaDA is a static deadlock analyzer that targets Java bytecode. The core of JaDA is a behavioral type system especially designed to record dependencies between concurrent code. These behavioral types are thereafter analyzed by means of a fixpoint algorithm that reports potential deadlocks in the original Java code. We give a practical presentation of JaDA, highlighting the main connections between the tool and the theory behind it and presenting some of the features for customising the analysis. We finally assess JaDA against the current state-of-the-art tools, including a commercial grade one.

8.1 Introduction

In concurrent languages, a *deadlock* is a circular dependency between a set of threads, each one waiting for an event produced by another thread in the set. In the Java programming language, deadlocks are usually *resource-related*, namely they are caused by operations ensuring different threads the exclusive access to a set of resources. (Java has also so-called *communication-related* deadlocks, which are common in network based systems. These deadlocks, which are thoroughly studied in [1,2], are out of the scope of this work.) Java features threads by means of an ad-hoc class called Thread; this class has two methods Thread.start() and Thread.join() for spawning and joining threads. The consistency between threads that share objects is enforced by *synchronized blocks*, a linguistic construct that may be defined either for

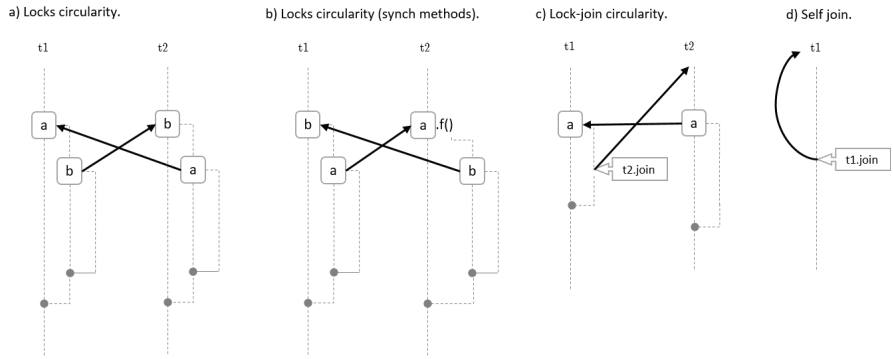

Figure 8.1 Cases of circular dependencies that may lead to deadlocks. (Lock acquisitions are represented with squares, the corresponding release is marked with a circle).

simple code blocks or for method bodies [3, Chapter 17][1]. It turns out that the dependencies defined by synchronized blocks may be circular. These problems are difficult to detect or anticipate, since they may not occur during every execution. Figure 8.1 shows (a timeline representation of) some examples of deadlocked programs. At the time of writing this chapter, the *Oracle Bug Database*[2] reports more than 40 unresolved bugs due to deadlocks, while the *Apache Issue Tracker*[3] reports around 400 unresolved deadlock bugs. Clearly, a deadlock may have catastrophic effects for the overall functionality of a software system.

In this chapter, we present an end-to-end automatic analyzer for detecting potential deadlock bugs of `Java` programs *at compilation time* – `JaDA`, the `Java` `Deadlock` `Analyzer` tool. `JaDA` addresses the compilation target of every `Java` application – the `Java` *Virtual Machine Language*, `JVML`, also called `Java` bytecode – and extracts abstract models out of it by means of an inference system. These abstract models are successively analyzed.

The decision of addressing `JVML` instead of `Java` was motivated by two reasons: `Java` is too complex and it has no reference semantics. On the contrary, `JVML` is simple – it has 198 instructions that can be sorted into 7

[1]There are also other mechanisms that remain out of the scope of this work, such as, the `volatile` variables and the higher-level synchronization API defined on package `java.util.concurrent`.

[2]http://bugs.java.com/

[3]https://issues.apache.org/jira

different groups of similar instructions – and has a reference semantics that is defined by the behavior of the Java *Virtual Machine* (JVM) [3, Chapter 6]. Analyzing JVML has also other relevant advantages: addressing programming languages that are compiled to the same bytecode, such as Scala [4], and the possibility to analyze proprietary software whose sources are not available.

The inference system of JaDA consists of a number of rules that analyze the effects of the instructions on the synchronization process. The types inferred from the bytecode, called *lams* [1, 2, 5, 6], are functional programs that define dependencies between threads. Then JaDA uses a variation of the algorithm defined in [1, 2] for detecting circularities in lams, and reports potential threats as output of the analysis. The tool also exhibits the executions causing deadlocks, by linking the dependencies with the chunk of source code that originated them, thus easing the analysis of false positives.

The current release of JaDA covers most of the JVML, including threads and synchronizations, constructors, arrays, exceptions, static members, interfaces, inheritance, recursive data types. Few synchronization-related features are not covered by the current release, such as `wait-notify-notifyAll` operations, dynamic class loading and reflection.

The rest of the chapter is organized as follows. Section 8.2 presents a motivating example of a recursive Java program that creates a (statically) unbounded number of threads. This is one of the main achievements so far and the theory overviewed in Section 8.3 will be explained by means of it. Section 8.4 describes the tool in some detail, highlighting implementation issues. Section 8.5 analyses the current limitations of JaDA and Section 8.6 reports an assessment of JaDA with respect to state-of-the-art tools for Java deadlock analysis. Finally we conclude in Section 8.7.

8.2 Example

Figure 8.2 reports the Java class `Network` and part of its JVML. The `main` method creates a network of `n` threads by invoking `buildNetwork` – say $t_1, \cdots, t_n$ – that are all potentially running in parallel with the caller – say t_0. Every two adjacent threads share an object, which is also created by `buildNetwork`.

The `buildNetwork` method will produce a deadlock depending on its actual arguments: it is deadlock-free when it is invoked with two different objects, otherwise it may deadlock (if also `n` $>$ 0). Therefore, in the

```
class Network{

 public void main(int n){
   Object x = new Object();
   Object y = new Object();

   // deadlock
   buildNetwork(n, x, x);

   // no deadlock
   //buildNetwork(n, x, y);
 }

 public void buildNetwork(int n,
                Object x, Object y){
   if (n==0) {
     takeLocks(x,y) ;
   } else {
     final Object z = new Object() ;

     //anonymous Thread child class
     Thread thr = new Thread(){
       public void run(){
         takeLocks(x,z) ;
     }} ;
     thr.start();
     this.buildNetwork(n-1,z,y) ;
   }
 }

 public void takeLocks(Object x,
                       Object y){
   synchronized (x) {
     synchronized (y) { }
   }
 }
}
```

```
public void buildNetwork(int n, Object x, Object y)
   0  iload_1          //n
   1  ifne 13
   4  aload_0          //this
   5  aload_2          //x
   6  aload_3          //y
   7  invokevirtual 24 //takeLocks(x, y):void
  10  goto 50
  13  new 3
  16  dup
  17  invokespecial 8  //Object()
  20  astore 4         //z
  22  new 26
  25  dup
  26  aload_0          //this
  27  aload_2          //x
  28  aload 4          //z
  30  invokespecial 28 //Network$1(this, x, z)
  33  astore 5         //thr
  35  aload 5          //thr
  37  invokevirtual 31 //start():void
  40  aload_0          //this
  41  iload_1          //n
  42  iconst_1
  43  isub
  44  aload 4          //z
  46  aload_3          //y
  47  invokevirtual 36 //buildNetwork(n-1, z, y):void
  50  return

public void takeLocks(Object x, Object y)
   0 aload_1;          //x
   1 dup;
   2 astore_3;
   3 monitorenter;     //acquires x
   4 aload_2;          //y
   5 dup;
   6 monitorenter;     //acquires y
   7 monitorexit;      //releases y
   8 aload_3;
   9 monitorexit;      //releases x
  16 return;
```

Figure 8.2 Java `Network` program and corresponding bytecode of methods `buildNetwork` and `takeLocks`. Comments in the bytecode give information of the objects used and/or methods invoked in each instruction.

case of Figure 8.2, the program is deadlocked, while it is deadlock free if we comment the instruction `buildNetwork(n,x,x)` and uncomment `buildNetwork(n,x,y)`.

The problematic issue of `Network` is that the number of threads is not known statically – `n` is an argument of `main`. This is displayed in the bytecode of `buildNetwork` in Figure 8.2 by the instructions at addresses 30 and 37 that

respectively created a new thread and start it, and by the recursive invocation at instruction 47.

8.3 Overview of JaDA's Theory

JaDA's theory relies on two main techniques: (*i*) an inference type system for extracting abstract models out of JVML instructions, and (*ii*) a fixpoint algorithm for analyzing the models. We overview the two techniques in the following subsections; in the last subsection we discuss the JaDA behavioral types for the buildNetwork example.

8.3.1 The Abstract Behavior of the Network Class

Figure 8.3 details the output of JaDA for the Network class in Figure 8.2. The types have been simplified for readability: the actual JaDA types are more complex and verbose. Some comments (in gray) explain the side effects of invocations, other comments (in yellow) correspond to the lines that are commented in Figure 8.2. The behavior of main begins by calling the constructor of the class Object. Notice that, after such invocation, the structure of x and y is known. Then the type reports the invocation to buildNetwork.

The behavior of takeLocks is the parallel composition of two dependencies corresponding to the acquisition of the locks of x and y. Every dependency is formed by the last held lock and the current element. Notice that every method receives an extra argument corresponding to the last acquired lock at the moment of the invocation, in this case that argument is u.

The behavior of buildNetwork has five states: (*i*) the invocation to takeLocks, (*ii*) the creation and initialization of the object z, (*iii*) the creation and initialization of the thread thr, (*iv*) the spawn of thr, (*v*) and the recursive invocation (in parallel with the spawn of thr). The buildNetwork method also reports one spawned thread as side effect. This may appear contradictory (because buildNetwork spawns n threads). However, in this case JaDA is able to detect that thr is the only thread (from those created) that may be relevant (for the deadlock analysis) in an outer scope. This deduction is done by considering the objects in the record structure of thr.

The constructors of Object and Thread have an empty behavior. On the contrary, the constructor of the class Network$1 is more complex (Network$1 is the name the JVM automatically gives to the anonymous

```
main(this | t,u):T{thr} =
    Object.init(x | t,u) + Object.init(y | t,u) +     //structure of x:x[] and y:y[]
    //deadlock
    buildNetwork(this,_,x,x | t,u)
    //no-deadlock
    //buildNetwork(this,_,x,y | t,u)                    //creates unsync thread: thr

takeLocks(this,x,y | t,u) = t:(u,x) & t:(x,y)

buildNetwork(this,_,x,y | t,u) :T{thr}=
    takeLocks(this,x,y | t,u) +
    Object.init(z | t,u) +                  //z:z[]
    Network$1.init(thr, this, x, z | t, z)+
     //thr:thr[this$0:this[], val$x:x[], val$z: z[]]
    Network$1.run(thr | thr,u1) +
    Network$1.run(thr | thr,u1) & buildNetwork(this,_,z,y | t,u)

Object.init(this | t, u):this[] = 0 //no side effects

Thread.init(this | t, u):this[] = 0 //no side effects

Network$1.init(this, x1, x2, x3 | t, u):this[this$0:x1, val$x:x2, val$z:x3] =
    Thread.init(this | t, u)

Network$1.run(this[this$0:x1,val$x:x2,val$z:x3]|t,u) = takeLocks(x1,x2,x3|t,u)
```

Figure 8.3 BuildNetwork's lams.

Thread child class[4] instantiated inside the method buildNetwork of the class Network). Being defined as an inner class, Network$1 has access to the local variables in the scope in which it has been created, namely the variables x, z and the this reference to the container instance. The JVM addresses this by passing these variables to the constructor of the class and assigning them to internal fields, in this case named valx, valz and this$0. Notice that the behavior of the constructor keeps track of two important things: the invocation to the constructor of the parent class Thread.init and the changes in the carrier object which goes from this to this[this$0:x1, val$x:x2, val$z:x3] where x_i are the formal arguments.

Finally, the behavior of the run method from the class Network$1 contains only the invocation to the takeLocks method. Notice that run method assumes a certain structure from the carrier object.

8.3.2 Behavioral Type Inference

The typing process is done bottom-up, in a compositional way. That is, a type is derived for every JVML instruction; the type of each method is the composition of the types of the instructions it contains. Similarly, the type of a program is the set of type of the methods therein. JaDA types are not standard types, such as integers, booleans, etc. They are models of the abstract behavior of a program, called *behavioral types*, that hold information about concurrency and synchronizations of every execution path.

In particular, the types of instructions retain two key pieces of information in JaDA:

- the *dependencies*, written t:(a,b), to be read as "thread t acquires the lock of object b while it is holding the lock of a", and
- the *method invocations*, written C.m($args$| t, a), which means that the method m of class C is invoked with arguments $args$ (that include the carrier object) in the thread t and while holding the lock of a.

In order to verify the consistency of parallel threads, behavioral types also take into account the (reading and writing) *effects* on objects. The types of the instructions can be composed either sequentially with the $+$ operation, or in parallel with the $\&$ operation.

In JaDA, the behaviors of methods are the sequential composition of instructions' types in method's body plus the sum of their effects. The effects

[4] https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

also include both threads spawns and thread joins in the method body. It is worth to remark that thread creations and synchronizations in `JVML` are defined by method invocations of the class `Thread`; therefore they are typed as method invocations with an ad-hoc management of spawns and joins.

The flow of the inference of behavioral types is described by the chart in Figure 8.4. The algorithm starts with an (empty) *Behavioral Class Table* (`BCT`), a structure where a behavioral description is associated to every method, and a sorted set of pending methods which initially contains all the methods of the program. The algorithm takes the first element of the set and types it (see below). The resulting effects are compared to the previous state of the `BCT`: if a change is found, the method is updated and every caller (every method depending on the current one) is added to the pending methods list. The algorithm terminates when the `BCT` reaches an stable state.

A similar technique is used to type the body of each method. The process is described in the chart shown in Figure 8.5. In this case, the inference process inside a method starts with a queue of pending instructions, which initially contains the first instruction. Each instruction is typed and the instruction *state* is updated (we have defined a set of typing rules in [7]). If the instruction type has been updated then the subsequent instruction(s) need to be typed (again). Notice that there may be several subsequent instructions, for example when the current instruction is a conditional. The state of an instruction contains an abstraction of the operand stack, the local variables, the local heap, the threads created upto that instruction and the chain of acquired locks

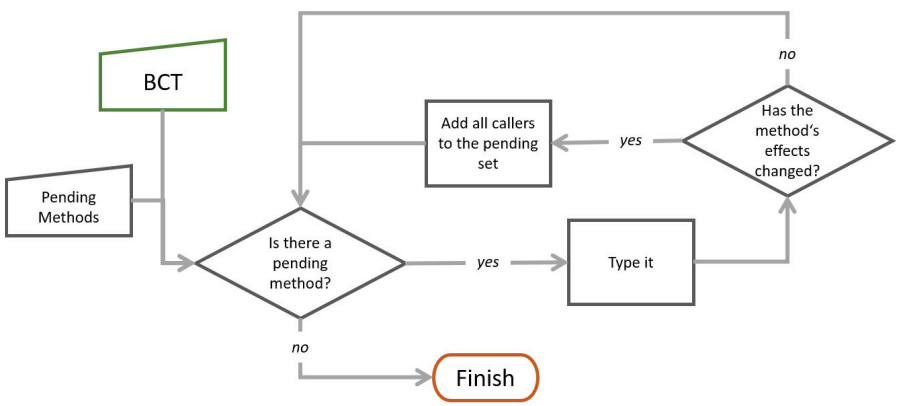

Figure 8.4 Type inference of methods' behaviors in `JaDA`.

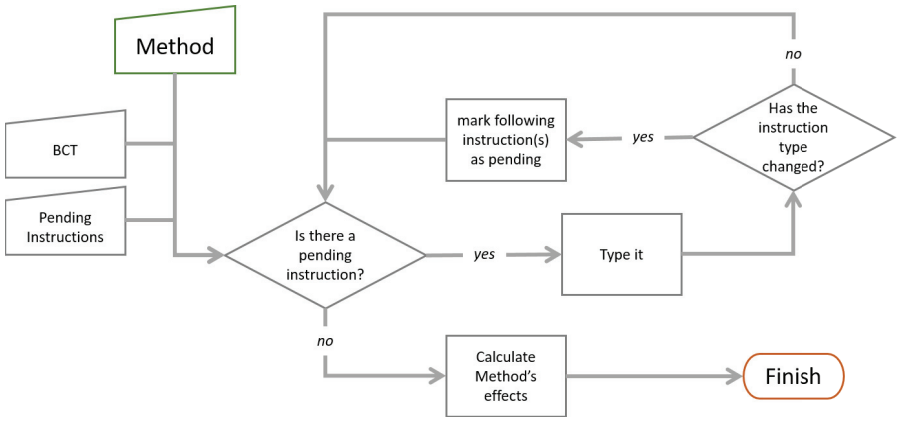

Figure 8.5 Type inference of method's bodies in `JaDA`.

(this information allows us to define the *lam* [2] of an instruction). Once no state is updated anymore, the type of the corresponding method is computed accordingly.

8.3.3 Analysis of Behavioral Types

The analysis of the inferred types is also performed iteratively. The overall approach is described by the chart in Figure 8.6. The initial step computes the *abstract state* of every method. This state is a sequence of parallel compositions of dependency pairs – function invocations in lams are deleted. The algorithm proceeds instruction by instruction, by *expanding* and *cleaning* its current state. The expansion process unfolds every invocation, the cleaning process removes pairs containing a fresh name (names not belonging to the method arguments or effects). Removing such pairs is crucial for termination because it allows us to keep the set of dependencies finite. In particular, the cleaning is performed by computing the transitive closure of the dependency pairs (this way we recover dependencies that are not direct and involve fresh names) and keeping only those whose elements are not fresh. In case we find a circular dependency formed only by fresh names then a special dependency pair is inserted (and this will ensure the presence of a deadlock). The full details of this algorithm are described in [1, 2].

Once all abstract states have been computed, the algorithm returns the circularities present in the `main` method.

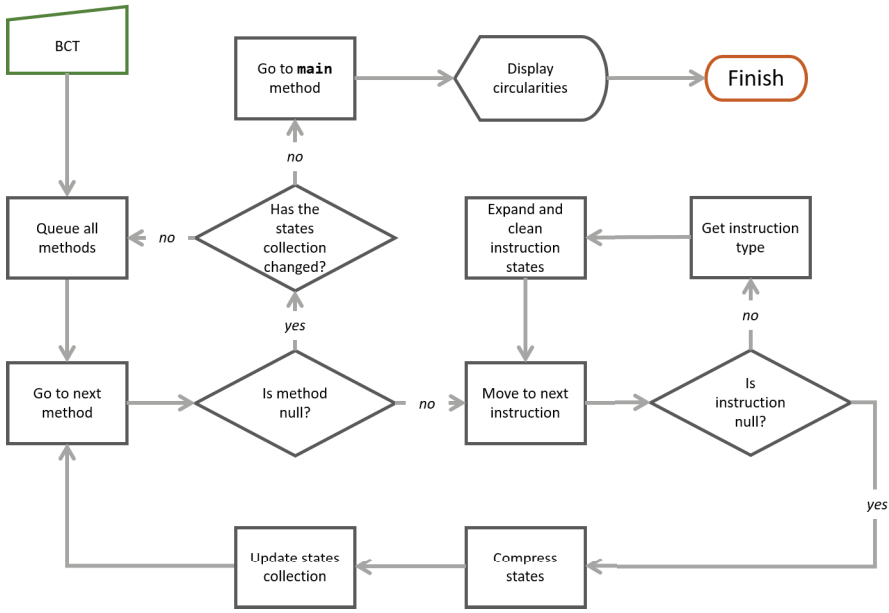

Figure 8.6 `JaDA` Analysis of behavioral types.

As an example, we apply the algorithm of Figure 8.6 to the `Network` behaviour in Figure 8.3. For simplicity we have excluded the methods with empty behaviour as well as their invocations.

Initially, an empty state is associated to every method. Using this model, we perform the first iteration and we get (we denote a set with `[ e`$_1$`, e`$_2$`, ···]` where elements `e`$_i$ are dependencies `t:(x,y)`; sets of sets are denoted by `[ [ e`$_1$`, e`$_2$`, ···], ···]`):

```
main(this | t,u) thr = [
        []   // no states resulting from buildNetwork(this,_,x,y | t,u)
] = []       // expanding and cleaning result empty

takeLocks(this,x,y | t,u) = [
        [ t:(u,x) & t:(x,y) ]
] = [
        [ t:(u,x) & t:(x,y) & t:(u,y)] // t:(u,y) is added by transitivity
]

buildNetwork(this,_,x,y | t,u)  thr = [
        [ t:(u,x) & t:(x,y) & t:(u,y)],  // invocation of takeLocks
        [],                    // invocation of Network$1.run
        [],                    // invocation of Network$1.run and buildNetwork
```

```
] = [
       [ t:(u,x) & t:(x,y) & t:(u,y) ]
]

Network$1.run(this[this$0:x1, val$x:x2, val$z:x3] | t, u) = [
       [ t:(u,x2) & t:(x2,x3) & t:(u,x3) ]      // invocation of takeLocks
] = [
       [ t:(u,x2) & t:(x2,x3) & t:(u,x3) ]
]
```

Since the states of methods is changed (all except `main`) we perform a second iteration, which gives:

```
main(this | t,u) thr = [
 [ t:(u,x) & t:(x,x) & t:(u,x)] // state of buildNetwork(this,_,x,y | t,u)
] =
[
 []  // the cleaning process removes dependencies that contain fresh names
     // the dependency t:(x,x) is removed because it is a reentrant lock
]

takeLocks(this,x,y | t,u) = [
 [ t:(u,x) & t:(x,y) ]
] = [
 [ t:(u,x) & t:(x,y) & t:(x,y)]  // this is the fixpoint for takeLocks
]

buildNetwork(this,_,x,y | t,u) thr = [
 [ t:(u,x) & t:(x,y) & t:(u,y)],           // invocation of takeLocks
 [ thr:(u,x) & thr:(x,z) & thr:(u,z)],     // invocation of Network$1.run
 [[ thr:(u,x) & thr:(x,z) & thr:(u,z)] & [ t:(u,z) & t:(z,y) & t:(u,y) ]]
                                // invocation of Network$1.run and buildNetwork
] = [
 [ t:(u,x) & t:(x,y) & t:(u,y)],  // this state has not changed
 [ t1:(u,x)],
 [ thr:(u,x) & t_thr:(x,y) & t:(u,y) ] // t_thr:(x,y) is new: it is a
             // dependency between x and y involving the threads t and thr
]

Network$1.run(this[this$0:x1, val$x:x2, val$z:x3] | t, u) = [
 [ t:(u,x2) & t:(x2,x3) & t:(u,x3) ] // invocation of takeLocks(x1,x2,x3 | t,u)
] = [
 [ t:(u,x2) & t:(x2,x3) & t:(u,x3) ] // this is the fixpoint for Network$1.run
]
```

Since `buildNetwork` is changed, we need a third iteration. The computation of the dependencies of `main` gives

```
main(this | t,u) thr = [// states resulting from buildNetwork(this,_,x,x | t,u)
        [ t:(u,x) & t:(x,x) & t:(u,x) ],
        [ thr:(u,x) ],
```

```
        [ thr:(u,x) & t_thr:(x,x) & t:(u,y) ]
] =
[
        [t_thr:($,$)]
]
```

In particular, in the states of `main`, after the transitive closure, contain `t_thr:(x,x)`, which is a circular dependency involving two threads. Instead of writing the dependency in that way (using a fresh name `x`), we write it as `t_thr:($,$)`, where `x` is replaced by a special name `$`. It is worth to notice that `t_thr:($,$)` gives two informations: (*i*) the deadlock is created by threads `t` and `thr`, (*ii*) the object name is `$`, which indicates that the deadlock is produced regardless of the arguments of the invocation. Since `t_thr:($,$)` denotes a circularity, the algorithm might stop. Actually, we decided not to stop `JaDA` at this point, we let it continue in order to collect every circularity.

`JaDA` output for the `Network` program is reported in Figure 8.7. In this case, `JaDA` has been set to analyze only the `Network` class (see *analysis-extent* in Section 8.4.4). Therefore, it warns about non-analyzed dependencies: the constructors from classes `Thread` and `Object` (whose types are considered empty – the actual type of these methods is nevertheless empty). `JaDA` reports 1 deadlock after the analysis, and outputs its trace. In this

```
[INFO] Analysis started at: 2016/12/08 09:37:55
[INFO]   Creating classpath for the analysis
[INFO]   Checking for classes located under: C:\JaDA\Git\Java-Deadlocks\[PaperExamples]\Network\bin
[INFO]   Analysis will run on the following target methods:
[INFO]     Network.main_(I)V
[INFO]   Calculating method behaviors
[WARNING] Method dependency not analyzed: java/lang/Thread.<init>
[WARNING] Method dependency not analyzed: java/lang/Object.<init>
[INFO]   Calculating method states
[INFO]   Method states calculation completed. Fixpoint process took 4 iterations.
[INFO]
[INFO]   NUMBER OF DEADLOCKS/LIVELOCKS: 1
[INFO]
[INFO]   1)
[INFO]   Deadlock found in method: main

       Thread 204 ($MAIN$) --this is the main thread-- tries to acquire:
main -> buildNetwork:30 -> buildNetwork:21 -> takeLocks:12 ->  x (346)
main -> buildNetwork:30 -> buildNetwork:21 -> takeLocks:12 ->  y (211)

       Thread 229 (thr) started at Network:15 (buildNetwork) tries to acquire:
main -> buildNetwork:30 -> run:15 -> takeLocks:17 ->  x (211)
main -> buildNetwork:30 -> run:15 -> takeLocks:17 ->  y (346)

[INFO]
[INFO]
[INFO] Analysis ended at: 2016/12/08 09:37:56. Analyis took 444 ms
```

Figure 8.7 `JaDA` analysis output for the `Network` program.

case there are two threads involved in the deadlock: those with id 204 (the one running `main`) and 229. The deadlock is caused by two `monitorenter` instructions on objects 346 and 211, taken in different order by the two threads. The tool outputs the computational traces ending with the two `monitorenter` instructions; the numbers in the traces represent the lines in the source[5].

8.4 The JaDA Tool

In this section we describe the main features of the `JaDA` tool, as well as, some key implementation details.

8.4.1 Prerequisites

`JaDA` has been designed to run on bytecode generated by the `Java` compiler[6] and it assumes that the bytecode has been already checked by the `Java` Bytecode Verifier (therefore it does not contain either syntactic or semantic errors). `JaDA` also requires that every dependency is matched by a corresponding bytecode. Although the bytecode is not executed, `JaDA` computes every necessary information to solve key issues for the analysis, such as the informations about inheritance. The loading of the existing types is done dynamically in a sand-boxed class loader[7] to avoid security risks. The full set of dependencies can be specified in `JaDA` through a *classpath*-like configuration (see property `class-path` in Section 8.4.4). Finally, `JaDA` also assumes that the code targeted by the analysis fits with the current limitations of the tool (see Section 8.5).

8.4.2 The Architecture

The `JaDA` analysis starts by parsing of the bytecode of a program and its dependencies. This is a cumbersome task because of the length and verbosity of the `JVML` syntax. `JaDA` relies on the `ASM` framework [8] for the bytecode extraction and manipulation. (Other third party tools have

[5]The line numbers in the output may not accurately match the example in Figure 8.2, because the latter has been slightly reduced for presentation purposes.

[6]We have tested `JaDA` against the 1.6, 1.7 and 1.8 versions of the `Java` compiler, and against the 1.8 version of the Eclipse Java Compiler (ECJ).

[7]https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html

been also designed for manipulating and analyzing the bytecode: the page `https://java-source.net/open-source/bytecode-libraries` contains a list of existing tools for this purpose. `ASM` provides a wide set of tools for interacting with the bytecode, including code generation and code analysis. It is also light-weight, open source, very well-documented and up todate with the latests versions of `Java`.

Figure 8.8 shows part of the `JaDA` architecture. In the figure, nodes are classes while arrows denote inheritance relationships. In the center of the image, there are the classes of the `ASM` framework; the other classes implement the technique so far described.

Values. A basic element of the architecture are the `Value` objects. `JaDA` uses two types of values: `RecordTree` store the methods' signature in the `BCT`, while `RecordPtr` store the state of local variables and the operand stack. Updating the `Value` elements amounts to upgrade every other element of the `JaDA` architecture. In the following paragraphs we discuss their functionality.

Frames. The `JDAFrame` class extends the `ASM` `Frame` by defining two important methods: `execute` and `merge`. The method `execute` implements the typing rules used by `JaDA`. It relies on an abstract interpreter that executes symbolically the current instruction with the given stack and local variables state. The method `merge` is invoked when the analysis process reiterates over an already typed frame. This method implements the logics of the *has changed* condition in the type inference of method bodies, see Section 8.3.2.

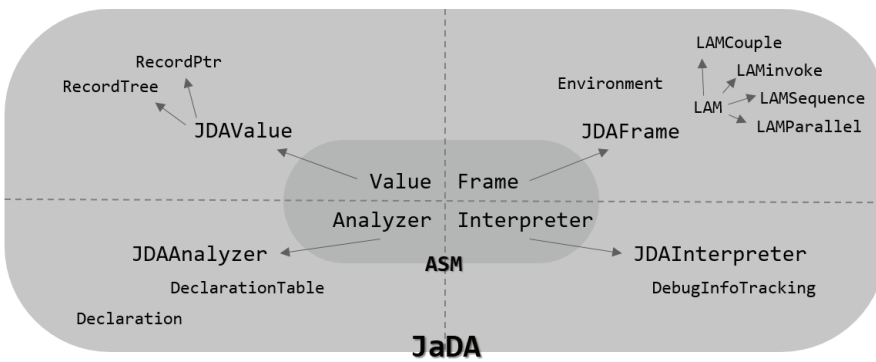

Figure 8.8 `JaDA` architecture.

```
                                20 ...
                                21 aload_1;          //a
                                22 dup;
                                23 astore 4;
                                25 monitorenter;
for(int i = 0; i < 10; i++){    26 aload_2;          //b
  synchronized (a) {            27 dup;
    synchronized (b) {          28 monitorenter;
    }                           29 monitorexit;
  }                             30 aload 4;
}                               32 monitorexit;
                                40 iinc 3 1;         //i++
                                43 iload_3;          //i
                                44 bipush 10;
                                46 if_icmplt -25;    //LOOP condition
                                48 ...;
```

Figure 8.9 Java `while` loop with nested synchronizations and the corresponding bytecode.

The decision on whether the subsequent frames must be checked again is taken upon the result of this method. To illustrate this consider the example from Figure 8.9.

When the typing process arrives each instruction for the first time its current Frame changes from the empty frame to the frame containing information about the instruction. Namely this first frame will contain the local variable status, the invocations and the existing locks and threads at each instruction. This changes enforces every frame to calculate its continuation at least one time. The following sequence shows the frames calculation for this chunk of code (only the relevant instructions are included):

```
...
Fr.21:{CurrentThread: main, Locks:{}}
...
Fr.25:{CurrentThread: main, Locks:{a}}
...
Fr.28:{CurrentThread: main, Locks:{a,b}}
Fr.29:{CurrentThread: main, Locks:{a}}
Fr.32:{CurrentThread: main, Locks:{}}
...
Fr.46:{CurrentThread: main, Locks:{}}
Fr.21:{CurrentThread: main, Locks:{}}
Fr.48:{CurrentThread: main, Locks:{}}
...
```

Notice that after calculating the frame 46, there are two possible continuations: 21 and 48. The second time Fr.21 is calculated it produces the same known result, therefore its continuation (Fr.22) is not calculated again. The calculation process continues then sequentially with Fr.48.

Interpreter. The JVM is a *stack machine*, every operation pops a certain number of elements off the stack and pushes on its result. The JDAInterpreter class extends the ASM Interpreter in order to comply with the values representations in JaDA. In particular, JDAInterpreter implements an important feature of our tool, namely the output of the traces potentially causing deadlocks. In fact, it returns the variable names of the objects involved, the stack trace chain and the related line numbers in the original Java code [8].

Analyzer. The ASM default analyzer supports very basic data-flow analysis limited to the scope of a single method. Similarly, JaDA analysis of a method does not go beyond its scope. However JDAAnalyzer extracts the necessary information – the type – that supports the compositional analysis. This part is implemented by the algorithm described in Figure 8.5, which is the building block of our tool. Consequently, JDAAnalyzer analyses the whole program by computing the final state of the BCT according to the algorithm in Figure 8.4. The final step of JDAAnalyzer is the computation of the models of the methods in the BCT as described in the algorithm in Figure 8.6.

8.4.3 The Current JVML Coverage

The theory of JaDA has been studied in [7] where we have defined the typing rules for a number of complex features of JVML, in particular, threads, synchronizations, static constructors, recursive data structures, inheritance and polymorphism, reflection and native methods. For a subset of this language – those featuring threads and synchronizations – we also delivered a correctness proof [9]. In this section we briefly overview our solutions for the main features of JVML that are covered in the current release of JaDA.

Static constructors. Static constructors are problematic because they are not explicitly invoked by the JVM. In fact, those are invoked on-the-fly by the JVM when the first (static) access to the containing class is performed. That is, the code of a static constructor can potentially precede any operation involving a static member of its class. In order to deal with this issue in a sound way, we model every static operation as a non-deterministic choice between the

[8] This is possible only when the bytecode has been compiled including debugging information.

type of the operation *per-se* and the typing of the static constructor of the class followed by the original operation. As one can imagine this makes the analysis computationally complex because the number of possible behaviors exponentially increases. The alternative (and the default choice) in `JaDA` is to assume that static constructors are all executed *before* the `main` method (see the `static-constructor` option in Section 8.4.4). This is a safe choice provided that concurrent operations do not occur within static constructors (which is often the case).

Recursive data structures. As discussed in Section 8.3.2, the analysis of `JaDA` relies on several iterative processes. The termination of these iterations strongly relies on the constraint that the number of object names is always finite. To ensure this finiteness constraint, the recursive objects are abstracted during the inference process. In particular, the inference replaces the field values whose class is already present in the object structure by a generic representative value. These representative values are treated in an ad-hoc way during the analysis of circular dependencies. Namely, they are considered equal to any other object of the same class (that is their identity is not guaranteed). Our assessments indicate that this over-approximation does not jeopardise `JaDA`'s precision when the elements of the recursive structure are pairwise different and threads act in a uniform way on them. On the contrary, the tool may return a number of false positives that is proportional to the dimension of the structure.

Arrays. Since `JaDA` does not process numerical expressions, it considers `array[2]` equal to `array[3]`. Therefore, `JaDA` manages arrays in a similar way it does for recursive data types. Every element in the array is represented by a unique object and, as for recursive data structures, this may be the cause of over-approximations. For example, `JaDA` returns a false positive when two different threads in parallel perform a lock operation on different objects of the array.

Inheritance and polymorphism. Inheritance and, in particular, polymorphism are sources of non-determinism. In fact, since it is not possible to resolve the runtime type of an object at static time, we cannot determine in a precise way the instance method being invoked over it. To deal with this issue in a sound way, `JaDA` substitutes every invocation with the non-deterministic choice among the method implementations in the type hierarchy

of the carrier. Enhancing this process to increase the precision of the analysis is currently an ongoing work. In the current release, whenever it is possible to derive the runtime type, we drop the wrong invocations.

Reflection, native methods, alternative concurrency models. In `Java`, like in many modern programming languages, there is some support for meta-programming, namely the capacity of a program to modify itself during its execution. `Java` also admits (native) methods and concurrency models that have no bytecode implementation. These methods are treated in an ad-hoc manner by the `JVM`. In all these cases, since there is not an explicit bytecode implementation, there is no evidence of what will happen at static time. Because of this reason, `JaDA` by default assumes a void behavior in these situations. Although, users can manually provide the behavior descriptions for methods involving such operations (see `custom-types` option in Section 8.4.4). This is particularly useful in the case of native methods (which are implemented in `C`), where users provide a more accurate behavior by analyzing its actual implementation.

8.4.4 Tool Configuration

In order to provide some flexibility, `JaDA` supports a set of settings to customize the analysis.

`<target>`**:** this setting specifies the target file or folder to analyze. It is mandatory. The type of files admitted are: `Java`class files (".class"), `Java` jar files (".jar") and compressed zip files (".zip"). In the case of folders, the content of the folder is analyzed recursively.

`verbose[=<value>]`**:** the value ranges from 1 to 5, the default and more verbose value is 5.

`class-path <classpath>`**:** Standard `Java` classpath description. If the target contains dependencies other than those in the standard library, they must be specified via this option.

`target-method <methodName>`**:** fully qualified target method (should be a void method without arguments). It compels `JaDA` to analyse the specified method. If this option is not set, the analysis chooses the first `main` method found.

`analysis-extent[=<value>]:` Indicates the extent of the analysis. Possible values are `full`: analyzes every dependency including the system and classpath-included libraries; `classpath`: analyzes every library in the classpath (this is the default value); `custom`: analyzes the classes specified through the property `additional-targets`; and `self`: does not analyze any class but the specified target.

`additional-targets <classes>:` if `analysis-extent` is set to `custom` this property must contain a comma separated list of the fully qualified names of a subset of classes in the *classpath* to include in the analysis. Such a feature is useful for avoiding typing known libraries.

`custom-types <file>:` a setting file to specify predefined behavioral types.

`static-constructors[=<value>]:` indicates when the static constructors should be processed, the possibilities are `before-all` and `non-deterministically`. The default option is `before-all`.

8.4.5 Deliverables

`JaDA` is available in three forms: a demo website [10], a command line tool (see Figure 8.7) and an Eclipse plug-in. All of them share the same core: a prototype implementation of the technique discussed in [7]. At the moment of writing this chapter, the demo website only allows to analyze single-file programs and to use a subset of the options previously described. The command line tool and the Eclipse plug-in are available through direct requests. The Eclipse plug-in output also displays the execution graph causing the deadlock with links to the source code that originates it (see Figure 8.10).

8.5 Current Limitations

The current version of `JaDA` does not cover a coordination mechanism between thread that is quite usual in `Java`: the `wait-notify-notifyAll` operations. There are also other less critical limitations, such as the analysis of native code and reflection operations. However, these features can be covered by manually specifying the behavior of the corresponding methods (see property `custom-types` in Section 8.4.4).

The methods `wait-notify-notifyAll` are public and final of the class `Object`; therefore they are inherited by all classes and cannot be modified.

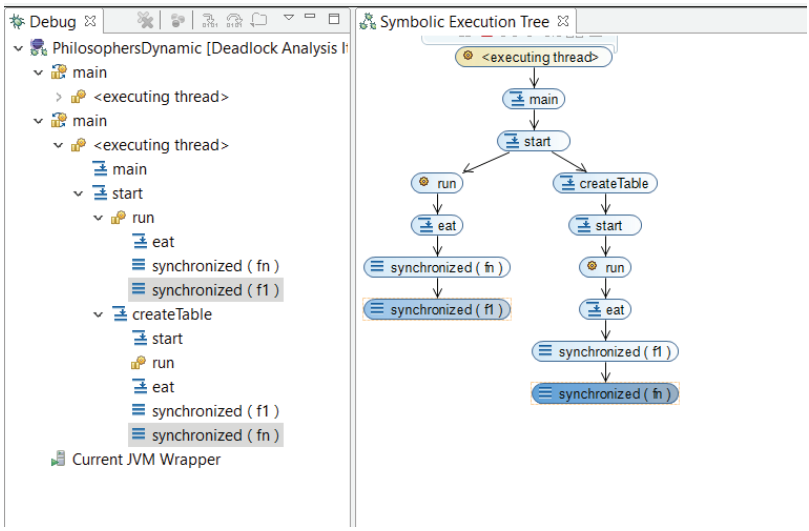

Figure 8.10 `JaDA` Eclipse plug-in screenshot.

The invocations to `wait`, `notify` and `notifyAll` succeed for threads that already hold the lock of the object a on the stack. In this case, the `wait` instruction moves its own thread to the *wait set* of a and the object is relinquished by performing as many unlock operations as the integer stored in the lock field of a. The instructions `notify` and `notifyAll` respectively wake up one thread and all the threads in the wait set of a. The woken-up threads are re-enabled for thread scheduling, which means competing for acquiring the lock of a again. The winner will lock a as many times it did on a before the `wait`-operation.

Below we briefly describe the solution we are currently investigating for extending `JaDA` to cover `wait-notify-notifyAll`.

We use two new type of dependency pairs: $t_1 : (a, a^n)$, which means "*the thread* t_1 *sends a notification on* a *while holding its lock*, and $t_2 : (a, a^w)$, which means "*the thread* t_2 *awaits a notification on* a *while holding its lock*". These two pairs are respectively produced by `notify` and `wait` methods. The problem is that, even if the abstract model retains a term $t_1 : (a, a^n) \& t_2 : (a, a^w)$ expressing that the `wait` and `notify` occur in parallel threads (*notification-wait* matching couple), we cannot conclude that the program is deadlock-free. This because the above term does not convey any information about what operation *has been performed before*. In fact, a wrong ordering might cause the thread t_2 to wait indefinitely. To overcome

this problem, we extend JaDA with an additional analysis that detects the *wait* pairs that can potentially remain unsatisfied. This solution is extensively discussed in [7].

8.6 Related Tools and Assessment

JaDA has been assessed with respect to a number of state-of-the-art tools. In particular, in Table 8.1, the tools have been classified according to the type of analysis they perform (see [7] for a discussion about analysis techniques for deadlock detection). We have chosen Chord for static analysis [11], Sherlock for dynamic analysis [12], and GoodLock for hybrid analysis [13]. We have also considered a commercial tool, ThreadSafe [9] [14].

We have analyzed a number of programs that exhibit a variety of sharing patterns. The source of all benchmarks in Table 8.1 is available either at [11, 12] or in the JaDA-deadlocks repository[10].

Since the current release of JaDA does not completely cover JVM, in order to gain preliminary experience, we modified the Java libraries and the multithreaded server programs of RayTracer, MolDyn and MonteCarlo (labelled with "(*)" in the Table 8.1) and implemented them in our system.

Table 8.1 Comparison with different deadlock detection tools. The inner cells show the number of deadlocks detected by each tool. The output labelled "(*)" are related to modified versions of the original programs: see the text

	Static		Hybrid	Dynamic	Commercial
Benchmarks	JaDA	Chord	GoodLock	Sherlock	ThreadSafe
Sor	1	1	7	1	4
RayTracer (*)	0	0	8	2	0
MolDyn (*)	0	0	6	1	0
MonteCarlo (*)	0	0	23	2	0
BuildNetwork	3	0			0
Philosophers2	1	0			1
PhilosophersN	3	0			0
StaticFields	1	1			1
ThreadArrays	1	1			1
ThreadArraysWJoins	1	1			0
ScalaSimpleDeadlock	1				
ScalaPhilosophersN	3				

[9] http://www.contemplateltd.com/threadsafe

[10] https://github.com/abelunibo/Java-Deadlocks

This required little programming overhead; in particular, we removed volatile variables, avoided the use of `Runnable` interfaces for creating threads, and reduced the invocations of native methods involved in I/O operations. Out of the four chosen tools, we were able to install and effectively test only two of them: `Chord` and `ThreadSafe`; the results corresponding to `GoodLock` and `Sherlock` come from [12] because we were not able to get the sources of the tools and run our new programs (*). We also had problems in testing `Chord` with some of the examples in the benchmarks, perhaps due to some misconfigurations, that we were not able to solve because `Chord` has been discontinued.

The first block of programs belongs to a well-known group used as benchmarks for several `Java` analysis tools. In its current state `JaDA` only detects 1 deadlock in all of the four analyzed programs from this group. It gives responses that are similar to `ThreadSafe` and `Chord` (`ThreadSafe` appears a bit more imprecise on Sor). The programs in the second block corresponds to examples designed to test our tool against complex deadlock scenarios like the `Network` program. We notice that both `Chord` and `ThreadSafe` fail to detect those kinds of deadlocks. The third group reports the analysis of two examples of `Scala` programs [4]. These programs have been compiled with the `Scala` compiler 2.11 whose target is `Java` bytecode. We remark that, to the best of our knowledge, at the moment of writing this chapter, there is no static deadlock analysis tools for such language (for this reason the entries corresponding to the other tools are empty).

We think that the results in Table 8.1 are encouraging and we hope to deliver more convincing ones as soon as `JaDA` overcomes its current limitations.

8.7 Conclusions

`JaDA` is a static deadlock analysis tool that targets `JVML`. Therefore it supports the analysis of every compiled `Java` program, as well as, every programs written in languages that are also compiled in `JVML`, like `Scala`. The technique underlying `JaDA` uses a behavioral type system that abstract the main features of the programs with respect to the concurrent operations.

`JaDA` is designed to run in an automatic fashion, meaning that the inference of the program type and the subsequent analysis could be done unassisted. Nevertheless, user intervention is possible and may enhance the precision of the analysis, for example in presence of native methods.

Even though the tool is still under development, we have been able to asses it by analyzing a set of `Java` and `Scala` programs. This contribution also reports a comparison between `JaDA`'s results and those of existing deadlock analysis tools, amongst which is a commercial grade one. The results obtained so far are very promising and we expect to gain more precision as the development continues.

References

[1] N. Kobayashi and C. Laneve, "Deadlock analysis of unbounded process networks," *Information and Computation*, vol. 252, pp. 48–70, 2017.

[2] E. Giachino, N. Kobayashi, and C. Laneve, "Deadlock analysis of unbounded process networks," in *Proceedings of 25th International Conference on Concurrency Theory CONCUR 2014*, vol. 8704 of *Lecture Notes in Computer Science*, pp. 63–77, Springer, 2014.

[3] J. Gosling, W. N. Joy, and G. L. S. Jr., *The Java Language Specification*. Addison-Wesley, 1996.

[4] M. Odersky and al., "An Overview of the Scala Programming Language," Tech. Rep. IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[5] E. Giachino and C. Laneve, "Deadlock detection in linear recursive programs," in *14th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014*, vol. 8483 of *Lecture Notes in Computer Science*, pp. 26–64, Springer, 2014.

[6] E. Giachino, C. Laneve, and M. Lienhardt, "A framework for deadlock detection in core ABS," *Software and Systems Modeling*, vol. 15, no. 4, pp. 1013–1048, 2016.

[7] A. Garcia and C. Laneve, "Deadlock detection of Java Bytecode." A preliminary version is available at `http://jada.cs.unibo.it/data/Doc/jada-draft-lncs.pdf`, 2016.

[8] E. Bruneton, "Asm 4.0 a java bytecode engineering library." `http://download.forge.objectweb.org/asm/asm4-guide.pdf`. Last accessed: 2016-12-03.

[9] A. Garcia, "Static analysis of concurrent programs based on behavioral type systems." Available at `http://jada.cs.unibo.it/data/Doc/Abel-Garcia-PhD-Thesis-draft.pdf`, 2017.

[10] A. Garcia and C. Laneve, "`JaDA` – the `Java` Deadlock `Analyzer`." Available at `http://jada.cs.unibo.it`, 2016.

[11] M. Naik, C. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *31st International Conference on Software Engineering (ICSE 2009)*, pp. 386–396, ACM, 2009.

[12] M. Eslamimehr and J. Palsberg, "Sherlock: scalable deadlock detection for concurrent programs," in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE-22)*, pp. 353–365, ACM, 2014.

[13] S. Bensalem and K. Havelund, "Dynamic deadlock analysis of multi-threaded programs," in *in Hardware and Software Verification and Testing*, vol. 3875 of *Lecture Notes in Computer Science*, pp. 208–223, Springer, 2005.

[14] R. Atkey and D. Sannella, "Threadsafe: Static analysis for java concurrency," *ECEASST*, vol. 72, 2015.