

Deadlock detection of Java Bytecode

Abel Garcia and Cosimo Laneve

Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus

Abstract. This paper presents a technique for deadlock detection of **Java** programs. The technique uses typing rules for extracting infinite-state abstract models of the dependencies among the components of the **Java** intermediate language – the **Java** bytecode. Models are subsequently analysed by means of an extension of a solver that we have defined for detecting deadlocks in process calculi. Our technique is complemented by a prototype verifier that also covers most of the **Java** features.

1 Introduction

Deadlocks are common flaws of concurrent programs that occur when a set of threads are blocked because each one is attempting to acquire a lock held by another one. Such errors are difficult to detect or anticipate, since they may not happen during every execution, and may have catastrophic effects for the overall functionality of the software system. At the time of writing this paper, the *Oracle Bug Database*¹ reports more than 40 unresolved bugs due to deadlocks, while the *Apache Issue Tracker*² reports around 400 unresolved deadlock bugs. These two databases refer to programs written in **Java**, a mainstream programming language in a lot of domains, such as web and cloud applications, user applications and mobile applications.

The objective of our research is to design and implement a technique capable of detecting potential deadlock bugs of **Java** programs *at static time*. This objective is difficult because **Java** has a complex concurrency model: it uses threads that may perform read/write operations over shared variables and whose execution depends on the scheduling strategy implemented in the Java Virtual Machine (**JVM**). In addition, **Java**, being a full-fledged programming language, includes an extensive standard library with lots of features implemented in native language.

To reduce the complexity of our work, we decided to address the **Java** bytecode, namely 198 instructions that are the compilation target of every **Java** application and have a reference semantics that is defined by the **JVM** behaviour. Therefore, it is possible to deliver correctness results without narrowing/oversimplifying our original goal. In this paper, we present our technique on a subset of **Java** bytecode, called **JVML_d**, which includes basic instructions for concurrency, such as thread creations, synchronizations, and creations of new objects. The language is defined in Section 3.

¹ <http://bugs.java.com/>

² <https://issues.apache.org/jira>

The technique consists of two stages. The first stage defines a type system that reconstructs the concurrent behaviour of methods. The key principles are the following ones. Each method has an associated type that depends on the type of the arguments (the object “**this**” is one argument) and that expresses the *concurrent behaviour*. This “concurrent behaviour” reports (i) the *sequence of locks* that has been acquired/released by the method, (ii) the *threads created*, and (iii) the *methods that have been invoked*. It includes the analysis of aliases that traces the creation of new objects and their copies (because JVM_d instructions may create and copy objects). The alias analysis is performed in a *symbolic way* by using a *finite set of names*: this is a critical part of our technique because methods may create threads and, when methods are either recursive or iterative, the set of created threads may be infinite. In particular, we had to devise finite representatives of (infinite sets of) thread names that are sound with respect to the (deadlock) analysis. Section 2 reports a code that can be written in (a simple extension of) JVM_d and that is problematic as regards deadlock detection. Section 4 describes the type system and Section 5 overviews the typing of complex features of JVM_L.

The second stage of our technique defines the analysis of the behavioural model. In fact, the three reports above – (i), (ii), and (iii) – are terms in a modelling language that extend so-called *lams* [7,6,9]. Lams are conjunctions and disjunctions of object dependencies and method invocations and the extension has been necessary for modelling **Java reentrant locks**. In particular, our dependencies also carry thread names – $(a, b)_t$ means that the thread t , which owns the lock of a , is going to lock b . In **Java**, the lam $(a, a)_t$ is not a circular dependency because it means that t is acquiring the same lock *twice*. Because of this extension, the algorithm for detecting circularities in lams is different than the one in [6,9]. We address this issue in Section 6.

Our deadlock detection technique has been prototyped and the verifier is called **JaDA**. While the type system in this paper simply checks static information, **JaDA** infers the behavioural types from the bytecode. Inference is important in practice because it lightens the analysis but checking is crucial for type safety³. **JaDA** includes several features of JVM_L; this has made possible to deliver initial assessments of the tool, which are discussed in Section 7. Section 8 discusses related work and reports our concluding remarks.

2 Overview of JVM_L and of our technique

Figure 1 reports a **Java** class called **Network** and some of its JVM_d representation. The corresponding **main** method creates a network of **n** threads – the philosophers – by invoking **buildNetwork** – say t_1, \dots, t_n – that are all potentially running in parallel with the caller – say t_0 . Every two adjacent philosophers share an object – the fork –, which is also created by **buildNetwork**. Every thread t_i locks the two adjacent forks, that are passed as (implicit) arguments of the thread, and terminates – this is performed by the method **takeForks**. It is well-known that when the network is a table (it is circular – the thread t_n is sharing one of its

³ The technical details of type safety appear in the full paper, where we also overview the inference system of **JaDA**.

```

class Network{
    public void main(int n){
        Object x = new Object();
        Object y = new Object();
        buildNetwork(n, x, y); //no deadlock
        buildNetwork(n, x, x); // deadlock
    }

    public void buildNetwork(int n,
        Object x, Object y){
        if (n==0) {
            takeForks(x,y) ;
        } else {
            final Object z = new Object() ;
            Thread t = new Thread(){
                public void run(){
                    takeForks(x,z) ;
                } ;
            t.start();
            this.buildNetwork(n-1,z,y) ;
        }
    }

    public void takeForks(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 //takeForks(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

```

Fig. 1. Java Network program and corresponding bytecode (only the `buildNetwork` method).

forks with t_0) and all the threads have a symmetric strategy of locking objects then a deadlock may occur. On the contrary, when either the network is not circular or one thread has an anti-symmetric strategy, no deadlock will ever occur. Therefore `buildNetwork(n,x,y)` is deadlock free, while `buildNetwork(n,x,x)` is deadlocked (when $n > 0$).

The problematic issue of **Network** is that the number of threads is not known statically because `n` is an argument of `main`. This is displayed in the bytecode of `buildNetwork` in Figure 1 by the instruction at address 30 where a new thread is created and by the instruction at address 37 where the thread is started. The recursive invocation that causes the (static) unboundedness is found at instruction 47. Our technique is powerful enough to cope with such problems and to predict the correct behaviour of the code of Figure 1 and the faulty one if we comment `buildNetwork(n,x,y)` and de-comment `buildNetwork(n,x,x)`. The technique works as follows. It infers abstract methods' behaviors by computing types, called *lams*, of their bytecode bodies. These lams abstract each bytecode instruction by dropping the *non-relevant* information for the deadlock analysis (e.g. operations on integer variables). In practice, the relevant operations for deadlock analysis are: locking operations (`monitorenter` and `monitorexit` instructions), thread spawning operations, function invocations and objects' structures. Thereafter the abstract model is analysed by a solver.

3 The language JVML_d

JVML_d is a restriction of JVML that includes basic constructs and instructions for concurrency⁴. In JVML_d, a program is a collection of *class files* whose methods have bodies written in JVML_d bytecode. This bytecode is a partial map from *addresses* ADDR to instructions. Addresses, ranged over by L, L', \dots , are intended to be natural numbers and we use the function $L + 1$ that returns the least address that is strictly greater than L . When P is a program, we write $\text{dom}(P)$ to refer to its domain (the set of addresses) and we assume that $0 \in \text{dom}(P)$ for every bytecode P .

We use a number of *names*: for classes, ranged over by C, D, \dots , for fields, ranged over by f, f', \dots , for methods, ranged over by m, m', \dots , and for local variables, ranged over by x, y, \dots . A possible empty sequence of names or syntactic categories of the following grammar is written by over-lining the name or the syntactic category, respectively. For instance a sequence of local variables is written \overline{x} . However, when we need to access to the elements of a sequence, we use the notation x_1, \dots, x_n . Class files CF are defined by the grammar:

$$\begin{aligned} CF &::= \text{class } C \{ \text{fields} : \overline{FD} \text{ methods} : \overline{MD} \} & MD &::= \top \text{ m } (C, \overline{T}) P \\ FD &::= C.f : T & T &::= \top \mid \text{int} \mid C \end{aligned}$$

where “*fields :*” and “*methods :*” are keywords and \top is a special type that include all the other types (any value of any type has also type \top). This type will represent values that are unusable in our static semantics. The type name C represents a class type, *which is never recursive* in JVML_d.

Instructions *Instr* of JVML_d bytecode are of the following form:

$$\begin{aligned} Instr &::= \text{inc} \mid \text{pop} \mid \text{push} \mid \text{load } x \mid \text{store } x \mid \text{if } L \mid \text{goto } L \\ &\quad \mid \text{new } C \mid \text{putfield } C.f : T \mid \text{getfield } C.f : T \mid \text{monitorenter} \mid \text{monitorexit} \\ &\quad \mid \text{invokevirtual } C.m(\overline{T}) \mid \text{start } C \mid \text{return} \end{aligned}$$

The informal meaning of these instructions is as follows:

- **inc** increments the content of the stack; **pop** and **push**, respectively, pops an element from the stack and pushes the integer 0 on the stack; **load** x and **store** x respectively loads the value of x on the stack and pops the top value of the stack by storing it in x ; **if** L pops the top value of the stack and either jumps to the instruction at address L , if it is nonzero, or goes to the next instruction; **goto** L is the unconditional jump;
- **new** C allocates a new object of type C , initializes it and pushes it on top of the stack; **putfield** $C.f : T$ pops the value on the stack and the underlying object value, and assigns the former to the field f of the latter; **getfield** $C.f : T$ pops the object on the stack and pushes the value in the field f of that object;
- **monitorenter**, **monitorexit** are the synchronization primitives that pop the object on the stack and lock and unlock it, respectively;

⁴ Actually, JVML_d has a minor difference with respect to JVML: in JVML, local variables are addressed by non-negative integers instead of names.

- **invokevirtual** $C.m(T_1, \dots, T_n)$ pops n values from the stack (the arguments of the invocation) and dispatches the method m on the object on top of the stack; when the method terminates, the returned value is pushed on the stack;
- **start** C creates and starts a new thread for the object on top of the stack. This operation corresponds to `invokevirtual java/lang/Thread/start()` on a thread of class C in JVM. We separate it from **invokevirtual** in order to provide more structure to our semantics (because it has an effect on the set of threads – see the operational semantics in the Appendix, where we also consider the instruction **join**);
- **return** terminates program execution.

The bytecode in Figure 1 is written in a sugared extension of JVM_d. In particular, **aload** and **iload** correspond to our **load** instruction (when the argument is an object or an integer, respectively), **ifne** corresponds to **if**, **dup** duplicates the top of the stack, **sub** subtracts the element on top of the stack from the last-by-one, **invokespecial** is the method invocation of the constructor of the class.

In order to simplify the presentation, in this paper we assume that fields are read-only as they cannot be modified after the initialisation (which is done by constructors that, in turn, are sequential) ⁵.

4 The type system

The purpose of the type system is to associate lams to JVM_d bytecodes. Since JVM_d is the target (of large part) of Java, the association is complex because we must deal with objects and aliasing, object creation and updates performed by constructors, and the concurrent operations – creation of new threads, lock and unlock operations. Therefore the details are pretty technical. In this paper we overview the type system by discussing features that are increasingly difficult. In particular, we will discuss one typing rule – that of **invokevirtual** – and we will study the basic, sequential case, the case of invocation of constructors, and the case of invocation of a concurrent thread. The complete set of rules appears in the full paper.

Typing rules associate types, which are *lams*, to JVM_d instructions by means of *judgments*. Typically, these judgments are abstractions of the machine states. In case of JVM, the state is a memory, called *heap* and the set of running *threads*. In turn, every thread is a stack of activation records – each one containing the address of the instruction to be performed, a stack, and a local memory – plus the sequence of locks owned. For example, in the case of **invokevirtual** (without arguments), the the element on the stack is the called object (which is used by the JVM to locate the right method body).

A possible judgment for the instruction at address i of the JVM_d program P is

$$\Gamma, F, S, Z, i \vdash_t P : \ell$$

where Γ , called *environment*, is the abstraction of the heap, F and S are the abstraction of the local memory and the stack, respectively, and Z is the sequence

⁵ The full paper reports the complete analysis that also addresses race conditions.

of locks acquired by the thread. The term ℓ is the *lam* of the instruction i and t is a symbolic name identifying the thread that is executing the instruction. (At static time it is not possible to model the stack of activation records).

Environments Γ , memories F , stacks S , and sequences of locks Z are defined by means of *types*, which are not lams. Types in judgments are more descriptive than those in JVML_d syntax; in particular object types are not just classes \mathbf{C} , that is records $[\mathbf{f}_1 : \mathbf{T}_1, \dots, \mathbf{f}_n : \mathbf{T}_n]$, where \mathbf{f}_i are the fields of the class. In fact, this notation is not adequate for dealing with aliasing. For example, let \mathbf{C} be a class with two fields \mathbf{f}_1 and \mathbf{f}_2 that store objects of class \mathbf{D} . If \mathbf{C} -objects are represented by $[\mathbf{f}_1 : \mathbf{D}, \mathbf{f}_2 : \mathbf{D}]$ then it is not possible to recover the identities of the values of \mathbf{f}_1 and \mathbf{f}_2 ; therefore we cannot distinguish the cases when \mathbf{f}_1 and \mathbf{f}_2 store the same object or two different objects, which is sensible when we compute object dependencies.

Therefore we decided to use *symbolic names*, ranged over by a, b, \dots , which also include *void* and *thread names* (threads are objects in **Java**; we use t, t', \dots when a name addresses a thread). Symbolic names allow us to define *flattened types* such as $[\mathbf{f}_1 : b, \mathbf{f}_2 : b]$ and $[\mathbf{f}_1 : b, \mathbf{f}_2 : c]$, thus separating the two foregoing cases. Actually, in order to avoid ambiguities with different classes having same field names, the flattened types also carry the class name, e.g. $([\mathbf{f}_1 : b, \mathbf{f}_2 : b], \mathbf{C})$.

The binding of symbolic names and flattened types is defined by the *environments*, ranged over by Γ, Γ_i, \dots . For example $[a \mapsto ([\mathbf{f} : b], \mathbf{C}), b \mapsto ([\mathbf{g} : \text{int}], \mathbf{D})]$ is an environment that defines the names a and b . The function $\text{typeof}(\Gamma, a)$ returns the type of a in Γ .

Finally, our type system uses *vectors* Γ, F, S, Z that are indexed by the addresses in $\text{dom}(P)$. The elements of these vectors are

- Γ_i is the environment at address i ;
- the map F_i maps local variables to type values;
- S_i is a sequence of type values;
- Z_i is the *sequence of symbolic names* locked at address i .

Simple methods. We begin with the rule for **invokevirtual** of a method that has no argument, does not modify the carrier and returns *void*:

$$\frac{\begin{array}{l} P[i] = \text{invokevirtual } \mathbf{C.m} () \quad i+1 \in \text{dom}(P) \\ S_i = a \cdot S' \quad \text{typeof}(\Gamma_i, a) = \mathbf{C} \\ \Gamma_{i+1} = \Gamma_i \quad S_{i+1} = \text{void} \cdot S' \quad F_{i+1} = F_i \quad Z_{i+1} = Z_i \end{array}}{\Gamma, F, S, Z, i \vdash_t P : \mathbf{C.m}(a, t, [Z_i])}$$

The rule verifies that the top element of the stack is of type \mathbf{C} and constraints the stack S_{i+1} to be the same as S_i , except for the top element, which is replaced by *void*. The lam of the instruction i indicates that the instruction is a method invocation: we will discuss this term later; we just notice that $[Z_i]$ is the first name in the sequence Z_i (that represents the last object locked by t).

Constructors. We continue by analysing methods that update the carrier, such as constructors, and return an object. It is usual in **Java** that the returned object is new (in the JVM it is a fresh run time name). In the type system we must be

careful with such names. Overall the set of symbolic names used by the type system must be finite, which is an issue when a new object is created inside an iteration or a recursion. To overcome this issue, we let symbolic name represent infinitely many instances of names in these cases. Technically, we use a function $names(i)$ that takes an address i and returns a tuple of names whose length is finite and depends on the address. This function returns a name that may occur already in the judgment when the instruction is not executed once.

The above rule has no element specifying (i) the type of the returned object and (ii) the number of fresh names created by the invocation. As regards (i), one could think to extend Γ with the typing of methods. Again, this is not adequate because of our need to trace the identity of objects. For example, let $C.m$ be a method that returns an object of class C ; we must distinguish the cases when $C.m$ is the identity or returns a new object with the same fields of the carrier, or with the two fields storing a same object, etc.

Therefore, in order to specify methods' types that are more informative than standard types, we decided to use a further map – the *behavioural class table*, noted BCT. The types used in the BCT are a variation of the above flattened types because we completely specify the tree structure of the object (BCT is a global map; it is not a vector of maps). These types are called *structured types* and are ranged over by ρ, ρ', \dots . For example

$$(a[f_1 : (b[g : \text{int}], D), f_2 : (c[g : \text{int}], D)], C)$$

is an object of class C whose symbolic name is a and that stores two different objects of class D in the fields. There is a simple way to transform a symbolic name and an environment into a structured type and conversely, to get an environment out of a structured type. We call these functions $mk_tree(\Gamma, a)$ and $env(\rho)$, respectively, and we leave their definitions as an exercise.

Let us discuss two examples of method types in the behavioural class table:

- $C.m$ is the identity; hence it returns the carrier and the type also specifies that the carrier has not been modified. The method type is

$$\text{BCT}(C.m) = (X, t, b) \rightarrow \langle X, X, \ell \rangle.$$

We notice that the type uses *variable names*, ranged over by X, Y, \dots , when the structure of the argument is not relevant. Additionally, the arguments of $C.m$ are three: the first element is the structured type of the carrier, the second and the third arguments are two symbolic names. The name t is the thread that performed the invocation and b is the last object name whose lock has been acquired by t . These two informations are used by the analyser to build the right dependencies between callers and callees and appear in the lam ℓ of the return type.

In the above method type, the carrier is addressed by X . This means that the symbolic name of the carrier is not used in the dependencies of ℓ . When this name is used, we write $\text{BCT}(C.m)$ as

$$((a[f_1 : X, f_2 : Y], C), t, b) \rightarrow \langle (a[f_1 : X, f_2 : Y], C), (a[f_1 : X, f_2 : Y], C), \ell \rangle,$$

which binds the occurrences of a in the return type.

- $\mathbf{C.p}$ is the constructor of the class \mathbf{C} that returns the carrier where the two fields have been initialised with the same new object of class \mathbf{D} (we assume \mathbf{D} has no field and we shorten $c[\]$ into c). In this case, $\text{BCT}(\mathbf{C.p})$ is

$$((a[\mathbf{f}_1 : X, \mathbf{f}_2 : Y], \mathbf{C}), t, b) \rightarrow (\nu c) \langle (a[\mathbf{f}_1 : (c, \mathbf{D}), \mathbf{f}_2 : (c, \mathbf{D})], \mathbf{C}), (a[\mathbf{f}_1 : (c, \mathbf{D}), \mathbf{f}_2 : (c, \mathbf{D})], \mathbf{C}), \ell \rangle$$

The relevant part of the return type is (νc) part. This part specifies that the name c is *new*, namely it does not occur in the arguments $(a[\mathbf{f}_1 : X, \mathbf{f}_2 : Y], \mathbf{C}), t, b$.

The last concept we need for presenting the new rule for **invokevirtual** is that of *instance* of a method type. An instance of $\text{BCT}(\mathbf{C.p})$ above when the arguments are $(a'[\mathbf{f}_1 : \top, \mathbf{f}_2 : \top], \mathbf{C}), t', b'$ (e.g. a' has been created without initialising the fields) is

$$\langle (a'[\mathbf{f}_1 : (c', \mathbf{D}), \mathbf{f}_2 : (c', \mathbf{D})], \mathbf{C}), (a'[\mathbf{f}_1 : (c', \mathbf{D}), \mathbf{f}_2 : (c', \mathbf{D})], \mathbf{C}), \ell\{a', t', b', c'/a, t, b, c\} \rangle.$$

This term will be written $\text{BCT}(\mathbf{C.p})((a'[\mathbf{f}_1 : \top, \mathbf{f}_2 : \top], \mathbf{C}), t', b')(c')$.

The type rule for a method $\mathbf{C.m}$ that updates the carrier (it is a constructor), has no argument and returns the updated carrier by creating one object is

$$\frac{\begin{array}{l} P[i] = \text{invokevirtual } \mathbf{C.m} () \quad i + 1 \in \text{dom}(P) \\ S_i = a \cdot S' \quad \text{typeof}(\Gamma_i, a) = \mathbf{C} \\ \rho = \text{mk_tree}(\Gamma_i, a) \quad b = \text{names}(i) \quad \text{BCT}(\mathbf{C.m})(\rho, t, [Z_i])(b) = \langle \rho', \rho'', \ell \rangle \\ \Gamma_{i+1} = \Gamma_i + \text{env}(\rho') + \text{env}(\rho'') \quad S_{i+1} = \text{root}(\rho') \cdot S' \quad F_{i+1} = F_i \quad Z_{i+1} = Z_i \end{array}}{\text{BCT}, \Gamma, F, S, Z, i \vdash_t P : \mathbf{C.m}(\rho, t, [Z_i]) \rightarrow \rho'}$$

The new part of this rule is the third line in the premise. In particular, in order to compute the instance of $\text{BCT}(\mathbf{C.m})$, we construct $\text{mk_tree}(\Gamma_i, a)$. The instance of the return type $\langle \rho', \rho'', \ell \rangle$ is used to update Γ_i (in this case $\rho' = \rho''$, therefore $\text{env}(\rho') = \text{env}(\rho'')$). The function $\text{root}(\rho)$ returns the root of the structured type ρ .

Concurrent methods. We finally discuss methods that are concurrent. Let $\mathbf{C.m}$ be a method that creates a new thread, say t' (and returns it). Since t' runs in parallel with the current thread, say t , the *conjunctive* effects of t and t' must be analysed by our tool (the second stage of our technique). To delegate the analyser to check the consistency of these conjunctive effects, the type system must record the threads that are created. To this aim, we extend our judgments with a set collecting such thread names. However, this set may be infinite (when the method is recursive or iterative). In order to have a more precise analysis, we distinguish the cases when the thread creation is executed once and those when the thread creation is executed several times. In the first case, the analyser will spawn exactly one thread; in the second case the analyser will spawn infinitely many threads (see the last part of the paragraph “*Lams*”).

As a consequence, our judgments have two sets of thread names: T for the names created once, R for the names that will be spawned infinitely many times, each time with a fresh name, and they become

$$\text{BCT}, \Gamma, F, S, Z, T, R, i \vdash_t P : \ell$$

We use the predicate “*i is executed once*” whenever the method containing the instruction i is not (mutual) recursive or the instruction i is not inside an iteration (this predicate can be easily computed in our type system). The type rule for a method $\mathbf{C.m}$ that creates two threads – t' executed once, t'' spawned several times – is

$$\begin{array}{c}
P[i] = \text{invokevirtual } \mathbf{C.m} \text{ ()} \quad i+1 \in \text{dom}(P) \\
S_i = a \cdot S' \quad \text{typeof}(\Gamma_i, a) = \mathbf{C} \\
\rho = \text{mk_tree}(\Gamma_i, a) \quad t', t'' = \text{names}(i) \quad \text{BCT}(\mathbf{C.m})(\rho, t, [Z_i])(t', t'') = \langle \rho', \{t'\}, \{t''\}, \rho'', \ell \rangle \\
\Gamma_{i+1} = \Gamma_i + \text{env}(\rho') + \text{env}(\rho'') \quad S_{i+1} = \text{root}(\rho') \cdot S' \quad F_{i+1} = F_i \quad Z_{i+1} = Z_i \\
T_{i+1}, R_{i+1} = \begin{cases} T_i \cup \{t'\}, R_i \cup \{t''\} & \text{if } i \text{ is executed once} \\ T_i, R_i \cup \{t', t''\} & \text{otherwise} \end{cases} \\
\hline
\text{BCT}, \Gamma, F, S, Z, T, R, i \vdash_t P : \mathbf{C.m}(\rho, t, [Z_i]) \rightarrow \rho'
\end{array}$$

In this case, the last premise defines the values of T_{i+1} and R_{i+1} according to the instruction i is executed once or not.

Lams. In our technique, the dependencies between symbolic names are expressed by means of *lams* [6], noted ℓ , whose syntax is

$$\ell ::= 0 \quad | \quad (a, b)_t \quad | \quad \mathbf{C.m}(\bar{\rho}) \rightarrow \rho' \quad | \quad (\nu a)\ell \quad | \quad \ell \&\ell \quad | \quad \ell + \ell$$

The term 0 is the empty type; $(a, b)_t$ specifies a dependency between the object a and the object b that has been created by the thread t . The term $\mathbf{C.m}(\bar{\rho}) \rightarrow \rho'$ defines the invocation of $\mathbf{C.m}$ with arguments $\bar{\rho}$ and with returned type ρ' . The argument sequence $\bar{\rho}$ has always at least three elements in our case: the first element is the carrier, while the last two elements are the thread that performed the invocation and the last object name whose lock has been acquired by it. The operation $(\nu a)\ell$ creates a new name a whose scope is the type ℓ ; the operations $\ell \&\ell'$ and $\ell + \ell'$ define the conjunction and disjunction of the dependencies in ℓ and ℓ' , respectively. The operators $+$ and $\&$ are associative and commutative.

A *lam program* is a pair (\mathcal{L}, ℓ) , where \mathcal{L} is a *finite set of function definitions*

$$\mathbf{C.m}(\bar{\rho}) \rightarrow \rho' = \ell_{\mathbf{C.m}}$$

with $\ell_{\mathbf{C.m}}$ being the *body* of $\mathbf{C.m}$, and ℓ is the *main lam*. We notice that the type ρ' is considered an argument of the lam function as well. When $\rho' = \text{void}$, the function definitions are shortened into $\mathbf{C.m}(\bar{\rho}) = \ell_{\mathbf{C.m}}$ and the invocations into $\mathbf{C.m}(\bar{\rho})$.

As an example, the lams of the **Network**'s code in Figure 1 is reported in Figure 2 (lams have been simplified for easing the readability). We discuss the methods **takeForks** and **buildNetwork**. The method **takeForks** has arguments **this**, **x**, **y**, **t** and **u**, where **t** and **u** are as discussed above. This method acquires the locks of **x** and **y** in order; therefore its lam is quite simple: there is a dependency between **u** and **x** and a dependency between **x** and **y**, namely $(\mathbf{u}, \mathbf{x})_t \& (\mathbf{x}, \mathbf{y})_t$. The lam of **buildNetwork** is more complex. The first line corresponds to the **then**-branch (lines 0-10), namely the invocation to **takeForks**. The other lines correspond to the **else**-branch. Here we have the creation of the object **z** and the invocation of the corresponding constructor (second line of the body of

```

Main(this,t,u) = (ν x,y)( Object.init(x,t,u)->x + Object.init(y,t,u)->y
                    + buildNetwork(this,_,x,y,t,u) )

takeForks(this,x,y,t,u) = (u,x)t & (x,y)t

buildNetwork(this,_,x,y,t,u) = (ν z,t1,u1)(
    takeForks(this,x,y,t,u)
    + Object.init(z,t,u) -> z
    + Network$1.init(t1[this$0:T,val$x:T,val$z:T],this,x,z,t,z) ->
      t1[this$0:this,val$x:x,val$z:z]
    + Network$1.run(t1[this$0:this,val$x:x,val$z:z],t1,u1) &
      buildNetwork(this,_,z,y,t,u)

Object.init(this,t,u) -> this = 0

Network$1.init(this[this$0:X,val$x:Y,val$z:Z],x1,x2,x3,t,u) ->
  this[this$0:x1,val$x:x2,val$z:x3] = 0

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

```

Fig. 2. Network's lams (the `_` is a place holder for an integer)

the lam function and line 17 of the bytecode), the invocation to the constructor of `Network`, that is called `Network$1`, which returns a new thread that we call `t1`. The last line of the lam of `buildNetwork` contains the invocation of `t1.start` and the recursive invocation to `buildNetwork`. These invocations are *in conjunction* because they are in parallel.

We conclude with a remark about the dependencies specified the the judgment

$$\text{BCT}, \Gamma, F, S, Z, T, R, i \vdash_t P : \ell.$$

In our system, these dependencies are actually those defined by ℓ and those defined by Z_i , T_i , and R_i . In particular, let $Z_i = a \cdot a'$, $T_i = \{t'\}$, and $R_i = \{t''\}$. Then the dependencies of the instruction i are $(mk_tree(\Gamma_i, t') = (t'[\overline{f : \rho'}], C)$ and $mk_tree(\Gamma_i, t'') = (t''[\overline{f : \rho''}], C)$:

$$\ell \& (a', a)_t \& \text{C.run}((t'[\overline{f : \rho'}], C), t', lock_{t'}) \& \text{RUN}(t''[\overline{f : \rho''}], C)$$

where C is a subclass of `Thread`, $lock_{t'}$ is a (fake) name associated to t' and representing a default object locked by t' , and `RUN` is a lam function defined by

$$\text{RUN}(a[\overline{f : \rho}], C) = \text{C.run}((a[\overline{f : \rho}], C), a, lock_a) \& (\nu a') \text{RUN}(a'[\overline{f : \rho}], C)$$

The difference between T and R is exactly the fact that `RUN` is recursive. This means that every name in R corresponds to the parallel composition of infinitely many threads with different root names. The analyser in Section 6 verifies whether this composition is consistent or not (with respect to deadlocks).

5 More about typing and JaDA

The type system described in this paper has been prototyped. It also covers features such as constructors, arrays, exceptions, static members, interfaces, inheritance, recursive data types. The overall system is called `JaDA`. Here we overview two relevant extensions – inheritance and recursive data types –, the details of these two extensions and the other ones can be found in Garcia's PhD thesis [5].

Inheritance. JVML_d does not admit to derive classes from other classes. As a consequence, when a method is invoked, it is possible to uniquely locate the method definition (the output of $\text{typeof}(\Gamma_i, a)$ in Section 4 *is always a single element*. Therefore we cannot type

```
C w ; { if (z) w = new D ; else w = new E ; } w.foo() ;
```

which is a correct Java program, assuming that D and E are subclasses of C. In this case, if D and E have different implementations of `foo`, we do not know how the invocation `w.foo()` will be dispatched at run-time. Our solution consists of relaxing the relation between consecutive environments Γ_i and Γ_{i+1} in such a way that the type of $\Gamma_{i+1}(w)$ may be the one of $\Gamma_i(w)$ *plus a set of subclasses therein*. Henceforth, the lam corresponding to the `invokevirtual` of `w.foo()` is $\sum_{C' \in \text{typeof}(\Gamma_i, w)} C'.\text{foo}(w, t, a)$, namely $C.\text{foo}(w, t, a) + D.\text{foo}(w, t, a) + E.\text{foo}(w, t, a)$.

Recursive types. Recursive types are managed by using finite representations. Object names of recursive types are *special names* indexed by $\$$. A flattened *recursive* record type is built by unfolding the recursive types (exactly) up to those nodes containing a name of a class already present in the tree. Nodes inside the tree are labelled by new names, nodes in the leaves are labelled either (for non recursive types) with \top or `int` or with names already present in the environment or (for recursive types) with names subscribed by a $\$$ that correspond to the nodes of the classes that are already present in the tree. By construction, these structures are finite. For instance, if C is a class whose type is `[val : Thread, next : C]` (a list of threads) then, in correspondence of a `new C` instruction, we produce an environment $r_\$ \mapsto [\text{val} : (a[], \text{Thread}), \text{next} : r_\$]$.

Lists like the foregoing one are managed in ad-hoc ways. In particular we can deliver a precise analysis as long as *the nodes of the list are all equal*, otherwise we return false positives. We observe that this technique is more precise than one would think. For instance, assume to create a list of threads, where the field `val` of each node contains a new thread. This list is created by an iteration and the instruction creating the thread is always the same – say i . Hence, by definition of the function $\text{names}(i)$, the nodes of the list always contain the same name and can be represented as described above. Finally, in order to have a sound analysis, we also modify our definition of circularity in lams. In particular, if the types of $r_\$$ and of $r'_\$$ are the same, a term like $(r_\$, c)_t \& (c, r'_\$)_{t'}$ is a circularity because $r_\$$ may be replaced by *every name of the same type*, including $r'_\$$.

6 The analysis of circularities in lams

Once behavioural types have been computed for the whole JVML_d program, we can analyse the type of the *main* method. The analysis uses an extension of the algorithm defined in [6,9] that we discuss below.

The semantics of lams is very simple: it amounts to unfolding function invocations. The critical points are that (i) every invocation may create new fresh names and (ii) the function definitions may be recursive. These two points imply that a lam model may have infinite states, which makes any analysis nontrivial. It is worth to recall that the states of lams are conjunctions ($\&$) of dependencies and function invocation (because types with disjunctions $+$ are modelled

by sets of states with conjunctive dependencies). The results of [6,9] allow us to reduce the analysis to *finite* models, *i.e.* finite disjunctions of finite conjunctions of dependencies. In turn, this finiteness makes possible to decide the presence of a so-called *circularities*, namely terms such as $(a, b)_t \& (b, a)_{t'}$.

In [6,9], the dependencies are not indexed by thread names: here we use more informative dependencies in order to cope with **Java** reentrant locks. In particular $(a, b)_t \& (b, a)_t$ is not a circularity and, when $t \neq t'$, we carefully separate it from $(a, b)_t \& (b, a)_{t'}$. Because of this extension, we have modified the definitions of *transitive closure* and of *projecting-out fresh names*, which are basic notions in the algorithm of [6,9]. Let $t \neq t'$ and let \checkmark be a special object name. This symbol \checkmark is a special thread name indicating that the dependency is due to the contributions of two or more threads. Let also ℓ be a conjunction $\&$ of dependencies:

- the *transitive closure* of ℓ , noted ℓ^+ , is the least conjunction that contains ℓ and such that if $(a, b)_t \& (b, c)_{t'}$ is a subterm of ℓ^+ then either (i) $(a, c)_{\checkmark}$ is a subterm of ℓ^+ , if $t \neq t'$, or (ii) $(a, c)_t$ is a subterm of ℓ^+ , if $t = t'$;
- ℓ has a *circularity* if there is a such that $(a, a)_{\checkmark}$ is a subterm of ℓ^+ .

For example $\ell = (a, b)_t \& (b, a)_t \& (b, c)_{t'}$ has no circularity because $\ell^+ = (a, b)_t \& (b, a)_t \& (a, a)_t \& (b, b)_t \& (b, c)_{t'} \& (a, c)_{\checkmark}$ does not contain any pair $(a, a)_{\checkmark}$.

As regards *projecting-out fresh names*, when a lam ℓ contains a function invocation, our algorithm replaces the invocation with the corresponding instance of its body *where new names are replaced by fresh names*. For example, if $\ell = (a, b)_t \& f(a, b)$ and $f(x, y) = (\nu z, t')((x, y)_{t'} \& (x, z)_{t'})$ then we obtain the term $\ell' = (a, b)_t \& (a, b)_{t'} \& (a, z')_{t'}$ – where z' is a fresh object name –, which is equal to its transitive closure. We notice that ℓ' may be simplified: (i) the dependency $(a, z')_{t'}$ will never be involved in a circular dependency because z' is fresh in t' and is unknown elsewhere; therefore it may be dropped; (ii) the dependency $(a, b)_{t'}$ is important, however the name t' is not: we just need to separate it from the other (old thread) names. Therefore we replace $(a, b)_{t'}$ with $(a, b)_{\bullet}$, where \bullet is a special thread name. The lam $(a, b)_t \& (a, b)_{\bullet}$ is the output of the projecting-out operation.

The algorithm we use is the following. Let (\mathcal{L}, ℓ) be a *lam program* and let \mathcal{L}_0 be the set of function definitions similar to \mathcal{L} but with bodies 0.

step 1: $i = 0$;

step 2: compute \mathcal{L}_{i+1} : for every body ℓ_f of a lam function f in \mathcal{L} compute the new body $\ell_f^{(i+1)}$ as follows:

- a. replace bound names with fresh names;
- b. replace function invocation in ℓ_f with their meaning in \mathcal{L}_i ;
- c. compute the transitive closure of the resulting lam and let ℓ_f^+ be the new lam;
- d. project out the fresh names in ℓ_f^+ and let $\ell_f^{(i+1)}$ be the resulting lam.

step 3: if $\mathcal{L}_{i+1} \neq \mathcal{L}_i$ then $i = i + 1$ and goto step 2, else exit.

The above algorithm terminates and let n be the least integer such that $\mathcal{L}_n = \mathcal{L}_{n+1}$. It turns out that ℓ will display a circularity (by evaluating its function invocations according to \mathcal{L} , which may be recursive) if and only if ℓ displays a

circularity when the function invocations are evaluated with their definitions in \mathcal{L}_n (which are not recursive).

The proof of soundness of our type system is represented by a subject reduction theorem expressing that, if a JVM configuration cn has lam ℓ and cn reduces to a configuration cn' then (i) cn' is also well-typed and (ii) if ℓ' is the type of cn' then, if a circularity occurs in ℓ' then a circularity is present in ℓ , as well.

7 Assessment of JaDA

Since JaDA covers many features of Java, it has been possible to deliver an initial assessment of it with respect to existing deadlock analysis tools. In particular, we have considered tools using different techniques **Chord** for static analysis [10], **Sherlock** for dynamic analysis [3], and **GoodLock** for hybrid analysis [2]. We have also considered a commercial tool, **ThreadSafe**⁶ [1]. Out of these 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 [3]. 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.

Table 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.

benchmarks	LOC, #Threads	deadlock	Static		Hybrid		Dynamic		Commercial	
			JaDA(tm)	Chord(tm)	GoodLock(tm)	Sherlock(tm)	ThreadSafe(tm)			
Sor	1274, 5	yes	1 [135s]	1 [210s]	7 [4s]	1 [39s]	4 [435s]			
RayTracer(*)	1292, 5	no	0 [155s]	0 [223s]	8 [2s]	2 [30s]	0 [502s]			
MolDyn (*)	1351, 5	no	0 [110s]	0 [191s]	6 [5s]	1 [49s]	0 [423s]			
MonteCarlo (*)	3619, 4	no	0 [231s]	0 [342s]	23 [5s]	2 [102s]	0 [821s]			
BuildNetworkN	40, N+1	yes	3 [8s]	0 [50s]			0 [50s]			
PhilosophersN	60, N+1	yes	3 [12s]	0 [51s]			0 [51s]			
ThreadArraysN	23, N+1	yes	1 [6s]	1 [40s]			1 [40s]			
ThreadArraysJoinsN	37, N+1	yes	1 [6s]	1 [41s]			0 [41s]			
ScalaSimpleDeadlock	39, 2	yes	1 [3s]							
ScalaPhilosophersN	62, N+1	yes	3 [4s]							

We have analysed a number of programs that exhibit a variety of sharing patterns. The source of all benchmarks in Table 1 is available either at [3,10] or in the **JaDA-deadlocks** repository⁷. 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 1) and implemented them in our system. 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. For every program, we give the lines of code (LOC), the number #Threads of threads explicitly created (in the second and third block this number depends on the argument N). We also state whether the program under examination has a deadlock

⁶ <http://www.contemplateld.com/threadsafe>

⁷ <https://github.com/abelunibo/Java-Deadlocks>

or not and the time in seconds (tm) each tool took to perform the analysis. The times for **GoodLock** and **Sherlock** were taken from the literature [3].

Here are our remarks. The first block of programs belongs to a well known group used as benchmarks for several **Java** analysis tools; the second block corresponds to examples designed to test **JaDA** against complex deadlock scenarios. First of all **JaDA** is the unique tool that never returns false positives or false negatives. **Chord** and **ThreadSafe** are unsound because they return false negative (see the second block). The execution time of the tools are similar (**JaDA** appears more efficient), except for **GoodLock** and **Sherlock**, which appear however much less precise (they return a lot of false positives). As regards the second block, we observe that **JaDA** returns few deadlocks, which do not depend from N . This is because our analysis is symbolic and does not consider numeric values (most of the deadlock are considered “to be similar”).

The third group reports the analysis of two examples of **Scala** programs [11] (the **Scala** compiler 2.11 produces **Java** bytecode). To the best of our knowledge, there is no static deadlock analysis tool for **Scala** (for this reason the entries corresponding to the other tools are empty).

We have also analyzed the whole **Java** library. The overall analysis took 5 hours and 40 min. We have considered as entry points the public static parameterless methods and we have run the analyzer with the following limitations: native codes are not analyzed (their behavioural type is 0) and concurrency dependencies caused by **wait/notify** patterns are not verified. The analysis has not reported any deadlock.

8 Related work and Conclusions

We do not have space to discuss in detail the related work; therefore we focus on the tools used in the assessment of Section 7 and their theories. **ThreadSafe** uses a data-flow analysis that constructs an execution flow graph and searches for cycles within this graph. Some heuristics are used to remove likely false positives. No alias analysis to resolve object identity across method calls is attempted. This analysis is performed in **Chord** [3,10], which can detect re-entrance on restricted cases, such as when lock expressions concern local variables (it is not possible to use fields). **GoodLock** [2] and its refinement **Sherlock** [3] use a theory that is based on monitors. Therefore the technique is a runtime technique that tags each segment of the program reached by the execution flow and specifies the exact order of lock acquisitions. Thereafter, these segments are analyzed for detecting potential deadlocks that might occur because of different scheduler choices (than the current one). This kind of technique is partial because one might overlook sensible patterns of methods’ arguments (*cf.* **BuildNetwork**, for instance). We are not aware of other static analysers for the deadlock detection of **Java**. A powerful static tool is **SACO** [4], which has been developed for **ABS**, an object-oriented language with a concurrent model different from **Java**. A comparison between **SACO** and a tool using a technique similar to the one in this paper can be found in [8].

In this paper we have defined a new technique for detecting deadlocks in **Java** programs by analysing the **Java** intermediate language **JVML**. The technique has been specified by focusing on a subset of **JVML** featuring thread creations and

synchronizations, called `JVMLd`. We have also developed a prototype, called `JaDA`, which also covers complex features of `Java`, such as static members, arrays, recursive data types, exception handling, inheritance and dynamic dispatch. These extensions have made possible to deliver an initial assessment of `JaDA` with respect to existing deadlock analysis tools for `Java`.

Our future work includes the analysis of features of `Java` that have not yet been studied. One relevant feature is thread coordination, which is expressed by the methods `wait`, `notify` and `notifyAll`. Another extension addresses *native methods*, namely methods that are not implemented within the language and that are used when it is necessary to interact with the Operating System or for meta-programming purposes. Our current solution is to manually insert in the BCT the behavioural types of native methods. We are investigating testing mechanisms that may help in writing the types of such methods.

References

1. Robert Atkey and Donald Sannella. Threadsafe: Static analysis for Java concurrency. *ECEASST*, 72, 2015. URL: <http://journal.ub.tu-berlin.de/eceasst/article/view/1025>.
2. Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Hardware and Software Verification and Testing*, volume 3875 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2005.
3. Mahdi Eslamimehr and Jens Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE-22)*, pages 353–365. ACM, 2014.
4. Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Proc. FORTE/FMOODS 2013*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2013.
5. Abel Garcia. *Static analysis of concurrent programs based on behavioral type systems*. PhD thesis, School in Computer Science and Engineering, 2017. Available at JaDA.cs.unibo.it.
6. Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of 25th International Conference on Concurrency Theory CONCUR 2014*, volume 8704 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2014.
7. Elena Giachino and Cosimo 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)*, volume 8483 of *Lecture Notes in Computer Science*, pages 26–64. Springer, 2014.
8. Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *Software and Systems Modeling*, 15(4):1013–1048, 2016.
9. Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017.
10. Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering (ICSE 2009)*, pages 386–396. ACM, 2009.
11. Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.