A Beginner's Guide to the DeadLock Analysis Model

Elena Giachino, Cosimo Laneve

A beginner's guide to the dead*L*ock *A*nalysis *M*odel

Elena Giachino and Cosimo Laneve

Dipartimento di Informatica, Università di Bologna – INRIA Focus Team
{giachino,laneve}@cs.unibo.it

Abstract. This paper is an introduction to the framework for the deadlock analysis of object-oriented languages we have defined in [6, 5]. We present a basic JAVA-like language and the deadlock analysis model in an accessible way. We also overview the algorithm for deciding deadlock-freeness by discussing a number of paradigmatic examples. We finally explore the techniques for coping with extensions of the object-oriented language.

Keywords: Static deadlock analyzers, object-oriented languages, circular dependencies, lam, livelocks.

1 Introduction

Modern systems are designed to support a high degree of parallelism by ensuring that as many system components as possible are operating concurrently. Deadlock represents an insidious and recurring threat when such systems also exhibit a high degree of resource and data sharing. In these systems, deadlocks arise as a consequence of exclusive resource access and circular wait for accessing resources. A standard example is when two processes are exclusively holding a different resource and are requesting access to the resource held by the other. An alternative description is that the correct termination of each of the two process activities *depends* on the termination of the other. Since there is a *circular dependency*, termination is not possible.

Deadlocks may be particularly insidious to detect in systems where the basic communication operation is asynchronous and the synchronization explicitly occurs when the value is strictly needed. Speaking a JAVA idiom, methods are *synchronized*, that is each method of the same object is executed in mutual exclusion, and method invocations are *asynchronous*, that is the caller thread continues its execution without waiting for the result of the called method. Additionally, an operation of *join* explicitly synchronizes caller and called methods (possibly returning the value of the called method). In this context, when a thread running on an object x performs a join operation on a thread on y then it blocks every other thread that is competing for the lock on x. This blocking situation corresponds to a dependency pair (x, y), meaning that the progress on x is possible provided the progress of threads on y. A deadlock then corresponds to a circular dependency in some configuration.

Further difficulties arise in the presence of infinite (mutual) recursion: consider, for instance, systems that create an unbounded number of processes such as server applications. In such systems, process interaction becomes complex and really hard to predict. In addition, deadlocks may not be detected during testing, and even if they are it can be difficult to reproduce them and find their causes.

Deadlock detection has been largely investigated both with static and run-time techniques [2, 10, 1] (just to cite few references). Static analysis guarantees that all executions of a program are deadlock-free, at the cost of being not precise because it may discard safe programs (false positives). Run-time checking cannot be exhaustive. However, whenever applicable, it produces fewer false positives than static analysis.

We adopt a static approach, while retaining the precision of a run-time checker in a large number of cases. Our deadlock detection framework consists of an inference algorithm that extracts abstract behavioral descriptions out of the concrete program. These abstract descriptions, called *lam programs*, an acronym for *deadLock Analysis Model*, retain necessary informations for the deadlock analysis (typically all the synchronization informations are extracted, while data values are ignored). Then a decision algorithm evaluates the abstract program for verifying its circularity-freeness (every state has no circular dependency). In turn, this property implies the deadlock-freeness of the original program. Our approach is then both *flexible* (since only the inference algorithm has to be adapted to the language, while the analyzer is language-independent) and *precise* (since the analyzer is a decision algorithm on a large class of lam programs – see below). The major benefits of our technique are that (i) it does not use any pre-defined partial order of resources and that (ii) it accounts for dynamic resource creation.

To overview our analyzer, we observe that, in presence of recursion in the code, the evaluation of the abstract description may end up into an infinite sequence of states, without giving back any answer. Instead, our analysis always terminates. The theoretical framework we have designed allow us to determine when to stop the evaluation. Informally, when the abstract program is *linear* – it has (mutual) recursions of the kind of the factorial function –, then states reached after some point are going to be equivalent to past states. That point, called *saturated state*, may be determined in a similar way as the orbit of a permutation [3] (actually, our theory builds on a generalization of the theory of permutations [6]). A saturated state represents the end of a pattern that is going to repeat indefinitely. Therefore it is useless to analyze it again and, if a deadlock has not been encountered up to that point, then it cannot be produced afterwords. Analogously, if a deadlock has been encountered, then a similar deadlock must be present each time the same pattern recurs. When the abstract program is nonlinear – it has (mutual) recursions of the kind of the fibonacci function – our technique is not precise because it introduce fake dependency pairs. However, it is sound.

The aim of this paper is to present our deadlock technique informally, by means of examples and without going into the (many) theoretical details. The interested reader may find the theoretical developments in [6] and the application of our abstract descriptions to a programming language in [5, 4].

The structure of the paper is as follows. In Section 2, we introduce the programming language and the analysis model. In Section 3, we overview the algorithm for detecting circularities in the analysis model. In Section 4, we discuss a number of programs and their associated abstract models. In Section 5, we explore two relevant extensions of the programming language: field assignment and an operation for releasing the lock. We conclude in Section 6.

2 Languages and models

2.1 The language `FJf`

The programs that are analyzed in this paper will be written in a JAVA-like language. Instead of using JAVA, which is quite verbose, we stick to a dialect of the ABS language [8], called `FJf`. To enhance readability, the semantics of `FJf` will be given in an indirect way by discussing the compilation patterns of the main constructs of `FJf` into JAVA (the reader is referred to [4] for a direct operational semantics).

`FJf` syntax uses four disjoint infinite sets of names: *class names*, ranged over by `A`, `B`, `C`, $\cdots$, *field names*, ranged over by `f`, `g`, $\cdots$, *method names*, ranged over by `m`, `n`, $\cdots$, and *variables*, ranged over by `x`, `y`, $\cdots$. The special name `this` is assumed to belong to the set of variables. The notation $\widetilde{\texttt{C}}$ is a shorthand for $\texttt{C}_1\,;\cdots;\texttt{C}_n$ and similarly for the other names. Sequences of pairs are abbreviated as $\texttt{C}_1\ \texttt{f}_1;\cdots;\texttt{C}_n\ \texttt{f}_n$ with $\widetilde{\texttt{C}}\ \widetilde{\texttt{f}}$. The syntactic categories of *class declarations* `CL`, *method declarations* `M`, *expressions* `e`, and *types* `T` are defined as follows

$$\begin{array}{lcl}
\texttt{CL} & ::= & \texttt{class C extends C}\ \{\widetilde{\texttt{T}}\ \widetilde{\texttt{f}}\ ;\ \widetilde{\texttt{M}}\} \\
\texttt{M} & ::= & \texttt{T m}\ (\widetilde{\texttt{T}}\ \widetilde{\texttt{x}})\{\ \texttt{return e}\ ;\} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e!m}(\widetilde{\texttt{e}}) \mid \texttt{new C}(\widetilde{\texttt{e}}) \mid \texttt{e;e} \mid \texttt{e.get} \\
\texttt{T} & ::= & \texttt{C} \mid \texttt{Fut(T)}
\end{array}$$

where sequences of field declarations $\widetilde{\texttt{T}}\ \widetilde{\texttt{f}}$, method declarations $\widetilde{\texttt{M}}$, and parameter declarations $\widetilde{\texttt{T}}\ \widetilde{\texttt{x}}$ are assumed to contain no duplicate names. A program is a pair (CT, `e`), where the *class table* CT is a finite mapping from class names to class declarations `CL` and `e` is an expression, called main expression.

According to the syntax, every class has a superclass declared with `extends`. To avoid circularities, we assume a distinguished class name `Object` with no field and method declarations and whose definition does not appear in the class table. We always omit the declaration "`extends Object`".

The main features of `FJf` are:

futures – `Fut(T)` these terms are called *futures* of type `T` and represent pointers to values of type `T` that may be not available yet. `Fut(T)` are the types of method invocations that have `T` as return type.

asynchronous method invocation – `x!m(y)`**:** the caller continues its execution when a method `m` of `x` is invoked without waiting for the result. The called method is evaluated in parallel (on a new spawned thread). The invocation `x!m(y)` has the same behavior as the following code written in Java

```
new Thread ( new Runnable() { public void run() { x.m(y); }
           }).start();
```

where, for the sake of conciseness, the code for exception handling is omitted. In particular, in Java, every `join()` statement and every synchronized method invocation should handle an `InterruptedException`.

mutual exclusion**:** in `FJf` every method invocation spawns a new thread. However at most one thread per object is executed at the same time. This is translated in Java by declaring every method to be `synchronized`.

thread synchronization for value retrieval**:** in `FJf` the caller retrieves the result of an invocation by the operation `t.get`, where `t` is a reference of the invocation. The `get` operation is blocking until the returned value is produced. The `FJf` code:

```
x!m(y).get;
```

corresponds to the synchronization behavior of the following code in Java (without exception handling management)

```
Thread t = new Thread ( new Runnable() {
                         public void run() { x.m(y); }
                       });
t.start();
t.join();
```

The operations `get` in `FJf` and `join` in Java have not the exact same meaning: while `get` synchronizes the two threads and retrieves the value, `join` is just a synchronization between the callee's and the caller's threads. However, from the point of view of deadlock analysis, the behaviors of the two operations are equivalent.

We omit examples at this stage: several `FJf` codes will be discussed in Section 4.

2.2 Analysis models for detecting circularities

In [6], we have developed a theoretical framework for defining relations on names (every pair of a relation defines a dependency between two tasks, the first one is waiting for the result of the second) and for determining whether a definition may produce a circular dependency – a *deadlock* – or not. The language for defining relation is called *lam* – an acronym for deadLock Analysis Model. Lams are defined by terms that use a set of *names*, ranged over by x, y, z, $\cdots$, and a disjoint set of *function names*, ranged over $\mathtt{m}$, $\mathtt{m}'$, $\mathtt{n}$, $\mathtt{n}'$, $\cdots$. A *lam program* is a tuple $\big(\mathtt{m}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{m}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\big)$ where $\mathtt{m}_i(\widetilde{x_i}) = \mathtt{L}_i$ are *function definitions* and $\mathtt{L}$ is the *main lam*. The syntax of $\mathtt{L}_i$ and $\mathtt{L}$ is

$$\mathtt{L} \quad ::= \quad \mathtt{0} \quad | \quad (x, y) \quad | \quad \mathtt{m}(\widetilde{x}) \quad | \quad \mathtt{L} \| \mathtt{L} \quad | \quad \mathtt{L} ; \mathtt{L}$$

such that (i) all function names occurring in $\mathtt{L}_i$ and $\mathtt{L}$ are defined, and (ii) the arity of function invocations matches that of the corresponding function definition.

It is possible to associate a lam function to each method of a `FJf` program. The purpose of the association is to abstract the object dependencies that a method will generate out of its definition. For instance, the `FJf` method declaration

```
C m1( C y, C z ) { return  ( y!m2( z ) ).get ; }
```

has the associated function declaration in a lam program

$$\mathtt{m1}(x, y, z) = (x, y) \| \mathtt{m2}(y, z)$$

where the first argument is the name of the object `this`, which is x for `m1` and y in the invocation of `m2`. The association method-definition/lam-function is defined by an inference system in [5]. In this paper, in order to be as simple as possible, we keep this association informal.

The semantics of a lam program requires a couple of preliminary notions. Let $\mathbb{V}, \mathbb{V}', \mathbb{I}, \cdots$ be partial orders on names (relations that are reflexive, antisymmetric, and transitive) and let $\mathbb{V} \oplus \widetilde{x} {<} \widetilde{z}$, with $\widetilde{x} \in \mathbb{V}$ and $\widetilde{z} \notin \mathbb{V}$, be the least partial order $\mathbb{V}'$ that satisfies the following rules

$$\mathbb{V} \subseteq \mathbb{V}' \qquad \frac{x \in \widetilde{x} \quad (x, y) \in \mathbb{V}' \quad z \in \widetilde{z}}{(y, z) \in \mathbb{V}'}$$

That is, $\widetilde{z}$ become *maximal names* of $\mathbb{V} \oplus \widetilde{x} {<} \widetilde{z}$. Let *lam contexts*, noted $\mathfrak{L}[\ ]$, be terms derived by the following syntax:

$$\mathfrak{L}[\ ] \quad ::= \quad [\ ] \quad | \quad \mathtt{L} \| \mathfrak{L}[\ ] \quad | \quad \mathtt{L} ; \mathfrak{L}[\ ]$$

As usual $\mathfrak{L}[\mathtt{L}]$ is the lam where the hole of $\mathfrak{L}[\ ]$ is replaced by $\mathtt{L}$. Finally, let $\mathit{var}(\mathtt{L})$ be the set of names occurring in $\mathtt{L}$.

The operational semantics of a program $\big(\mathtt{m}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{m}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}_{\ell+1}\big)$ is defined by a *transition relation* between *states* that are pairs $\langle \mathbb{V},\ \mathtt{L} \rangle$ and satisfying the rule:

$$\text{(Red)}\quad \frac{\mathtt{m}(\widetilde{x}) = \mathtt{L} \qquad \mathit{var}(\mathtt{L}) \setminus \widetilde{x} = \widetilde{z} \qquad \widetilde{w} \text{ are fresh} \qquad \mathtt{L}[{}^{\widetilde{w}}/_{\widetilde{z}}][{}^{\widetilde{u}}/_{\widetilde{x}}] = \mathtt{L}'}{\big\langle \mathbb{V},\ \mathfrak{L}[\mathtt{m}(\widetilde{u})] \big\rangle \longrightarrow \big\langle \mathbb{V} \oplus \widetilde{u} {<} \widetilde{w},\ \mathfrak{L}[\mathtt{L}'] \big\rangle}$$

By (Red), a lam is evaluated by successively replacing function invocations with the corresponding lam instances. At every evaluation step, free names in a lam definition $\mathtt{m}(\widetilde{u}) = \mathtt{L}$, namely $var(\mathtt{L}) \setminus \widetilde{x}$, are replaced by *fresh names*. This replacement models name creation and correspond to the `new` operation in `FJf`. For example, if $\mathtt{m}(x) = (x, y)\|\mathtt{m}(y)$ and $\mathtt{m}(u)$ occurs in the main lam, then $\mathtt{m}(u)$ is replaced by $(u, v)\|\mathtt{m}(v)$, where v is a *fresh maximal name* in some partial order. The initial state of a program with main lam $\mathtt{L}$ is $\langle \mathbb{I}_{\widetilde{x}},\ \mathtt{L}\rangle$, where $\widetilde{x} = x_1, \cdots, x_n = var(\mathtt{L})$ and $\mathbb{I}_{\widetilde{x}} = \{(x, x) \mid x \in \widetilde{x}\}$ (we are abusing of the set-notation).

Lams record sets of relations on names. To make explicit these relations, let $\flat(\cdot)$, called *flattening*, be the function inductively defined as follows

$$\flat(\mathtt{0}) = \mathtt{0}, \qquad \flat((x, y)) = (x, y), \qquad \flat(\mathtt{m}(\widetilde{x})) = \mathtt{0},$$
$$\flat(\mathtt{L}\|\mathtt{L}') = \flat(\mathtt{L})\|\flat(\mathtt{L}'), \qquad \flat(\mathtt{L}\,;\,\mathtt{L}') = \flat(\mathtt{L})\,;\,\flat(\mathtt{L}').$$

For example

$$\flat(\mathtt{m}(x, y, z)\,;\,(x, y)\|\mathtt{n}(y, z)\|\mathtt{m}(u, y, z)\,;\,\mathtt{n}(u, v)\|(u, v)\,;\,(v, u)) = (x, y)\,;\,(u, v)\,;\,(v, u)$$

that is, the argument of $\flat(\cdot)$ defines three relations: $\{(x, y)\}$ and $\{(u, v)\}$ and $\{(v, u)\}$. It is easy to verify that, up-to the axioms

$$(x, y)\|(x, y) = (x, y) \qquad (\mathtt{L}\,;\,\mathtt{L}')\|(x, y) = \mathtt{L}\|(x, y)\,;\,\mathtt{L}'\|(x, y)$$

$$\frac{\mathtt{L} = (x_1, y_1)\| \cdots \|(x_n, y_n)}{\mathtt{L}\,;\,\mathtt{L} = \mathtt{L}}$$

$\flat(\mathtt{L})$ always returns *sequences of pairwise different parallels of pairs* (i.e. sets of pairwise different relations).

Definition 1. *A lam* $\mathtt{L}$ has a circularity *if*

$$\flat(\mathtt{L}) = (x_1, x_2)\|(x_2, x_3)\| \cdots \|(x_m, x_1)\|\mathtt{L}'\,;\,\mathtt{L}''$$

for some $x_1, \cdots, x_m$. A state $\langle \mathbb{V},\ \mathtt{L}\rangle$ has a circularity if $\mathtt{L}$ has a circularity. A program $(\mathtt{m}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots,\ \mathtt{m}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L})$ is circularity-free *if no state yielded by evaluating $\langle \mathbb{I}_{var(\mathtt{L})},\ \mathtt{L}\rangle$ has a circularity.*

3 The algorithm for deciding circularity-freeness

In case of non-recursive lam programs, since the evaluations always terminate, the circularity-freeness problem is (easily) decidable. Otherwise – when functions are (mutually) recursive – the evaluation would not terminate and would produce infinite relations due to the creation of names. Nevertheless, the problem of circularity-freeness is decidable for a large set of mutual recursive lam programs – the linear ones.

Definition 2. *A lam program $(\mathtt{m}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{m}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L})$ is* linear *if, for every function $\mathtt{m}_{i_0}$, there is at most one sequence $\mathtt{m}_{i_0} \cdots \mathtt{m}_{i_m}$ such that, for every $0 \leq j \leq m$, $\mathtt{L}_{i_j}$ contains exactly one invocation of $\mathtt{m}_{i_{j+1\%m}}$ (the operation $\%$ is the remainder of the natural division).*

For example, a factorial-like programs, such as $\big(\texttt{fact}(x) = (x,y)\|\texttt{fact}(y),\ \texttt{fact}(x)\big)$, are linear, while fibonacci-like ones, such as $\big(\texttt{fib}(x) = (x,y)\|(x,z)\|\texttt{fib}(y)\|\texttt{fib}(z),\ \texttt{fib}(x)\big)$, are not.

The idea of our technique is to recognize the pattern of the recursion in order to be able to determine the states when the evaluation flow is going to repeat the same pattern (with different names) and only produce pattern of dependencies that were already discovered in the past of the evaluation. Once our algorithm recognizes these states, called *saturated states*, it just interrupts avoiding non-terminating evaluations.

Theorem 1 ([6]). *The problem of circularity-freeness in linear lam programs is decidable.*

The theoretical details of this theorem are out of the scope of this paper. Here we illustrate the algorithm on the following sample program: $\big(\texttt{fact}(x) = (x,y)\|\texttt{fact}(y),\ \texttt{fact}(x)\big)$. In this case, our theory affirms that the saturated state is reached in two steps of evaluation. That is

$$\begin{aligned}\big\langle \mathbb{I}_x,\ \texttt{fact}(x)\big\rangle &\longrightarrow \big\langle \mathbb{I}_x \oplus (x{<}y),\ (x,y)\|\texttt{fact}(y)\big\rangle \\ &\longrightarrow \big\langle \mathbb{I}_x \oplus (x{<}y) \oplus (y{<}z),\ (x,y)\|(y,z)\|\texttt{fact}(z)\big\rangle\ .\end{aligned}$$

We observe that, if we perform a further step of evaluation, we get

$$\big\langle \mathbb{I}_x \oplus (x{<}y) \oplus (y{<}z) \oplus (z{<}u),\ (x,y)\|(y,z)\|(z,u)\|\texttt{fact}(u)\big\rangle$$

and there is an injective partial map ρ on $\mathbb{I}_x \oplus (x{<}y) \oplus (y{<}z) \oplus (z{<}u)$, namely $\rho = [z \mapsto x, u \mapsto y]$, such that

1. $\texttt{fact}(u)$ is mapped to an invocation that is already evaluated – that is $\texttt{fact}(x)$;
2. dependencies produced by $\texttt{fact}(z)$, namely (z,u) are mapped to dependencies that have been already produced, namely (x,y).

These properties allow us to decide that the evaluation is repeating the same pattern (on new names) and conclude that no circularity will be ever manifested since the saturated state is circularity-free.

When the lam program is nonlinear, our technique is imprecise but sound: the nonlinear lam program is transformed into a linear one *by contracting* multiple method invocations to one. This contraction *introduces fake dependencies* (i.e. false positives in terms of circularities). Once the linear transformed program is obtained, then the analysis is run on it. It turns out that these additional dependencies cannot be eliminated because of a cardinality argument. More precisely, the evaluation of a method invocation $\texttt{m}(\widetilde{u})$ in a linear program may produce at most one invocation of $\texttt{m}$, while an invocation of $\texttt{m}(\widetilde{u})$ in a nonlinear program may produce two or more invocations of $\texttt{m}$. When the invocations of $\texttt{m}$ create names, *contracting* different invocations into one means reducing an exponential number of new names to a linear number. This, in terms of the analysis means losing precision. Nevertheless, we prove the soundness of our technique:

if the transformed linear program is circularity-free then the original nonlinear one is also circularity-free. For example, consider the fibonacci-like program

$$\big(\ \mathtt{fib}(x) = (x,y) \| (x,z) \| \mathtt{fib}(y) \| \mathtt{fib}(z),\ \mathtt{fib}(x)\ \big)$$

The transformed program is

$$\big(\ \mathtt{fib}^{aux}(x,x') = (x,y) \| (x,z) \| (x',y) \| (x',z) \| \mathtt{fib}^{aux}(y,z)\ , \\ \mathtt{fib}^{aux}(x,x)\big)$$

which is linear but adds dependencies. In this transformation, the two invocations of `fib` have been contracted into one – the $\mathtt{fib}^{aux}$ invocation – that carries two arguments, one for every invocation – notice that the invocation $\mathtt{fib}^{aux}(\mathtt{y},\mathtt{z})$ corresponds to the two invocations `fib(y)` and `fib(z)`. In the body of $\mathtt{fib}^{aux}$, the creation of the two names in `fib` is simulated by creating two names as well. However, these two names *are the same* both for the first argument and for the second one (that correspond to the two recursive invocations of `fib`). This results into fake dependencies between names originally belonging to two different and independent invocations. In particular, after two steps of evaluation, $\mathtt{fib}^{aux}(x,x)$ gives,

$$(x,y) \| (x,z) \| (y,y') \| (y,z') \| (z,y') \| (z,z') \| \mathtt{fib}^{aux}(y',z')$$

whilst the corresponding lam of the original nonlinear program is

$$(x,y) \| (x,z) \| (y,y') \| (y,y'') \| (z,z') \| (z,z'') \| \mathtt{fib}(y') \| \mathtt{fib}(y'') \| \mathtt{fib}(z') \| \mathtt{fib}(z'')\ .$$

We demonstrate that, if no circularity is manifested by the saturated state of $\mathtt{fib}^{aux}$ then the original fibonacci program is circularity-free. Since the circularity-freeness of $\mathtt{fib}^{aux}$ is decidable, by Theorem 1, then, in case of circularity-freeness, we are able to state the same property for `fib`.

4 The technique by examples

We illustrate our analysis technique by discussing a number of programs written in `FJf`. Every example is discussed as follows:

1. we first present the `FJf` description;
2. then we give the associated lam, by keeping informal the association technique (see [5] for on a inference system defining the formal association);
3. we finally inspect and evaluate lams looking for circularities.

Getting started: a simple deadlocked program. Consider the following class `C` with three synchronized methods `m1`, `m2`, and `m3`:

```
class C {
     C m1( C y, C z ) { return y!m2(z).get ; }
     C m2( C z ){ return z!m3( ).get ; }
     C m3( ) { return new C( ) ; }
}
```

An invocation `x!m1(y,z)` spawns a thread (in the thread pool of `x`) that will invoke `y!m2(z)` and immediately blocks because of the `get` (hence it will not release the lock on `x`), waiting for the value that is returned by `m2`. The invocation `y!m2(z)`, in turn, causes a new thread to be spawned (in the thread pool of `y`) that invokes `z!m3()` and blocks waiting for its result. The lam functions associated to the above methods are

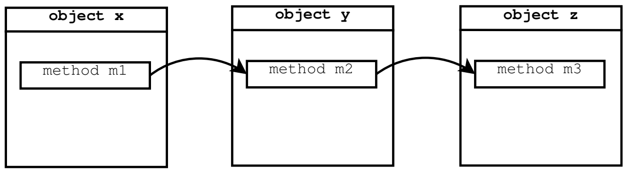
$$\begin{aligned}\mathtt{m1}(x,y,z) = &\ (x,y)\|\mathtt{m2}(y,z)\\ \mathtt{m2}(x,y) = &\ (x,y)\|\mathtt{m3}(y)\\ \mathtt{m3}(x) = &\ \mathtt{0}\end{aligned}$$

That is, to every method declaration we associate a lam declaration with an additional first argument representing the object `this` (which is always x). The body of the lam declaration ignores every local computation and translates `FJf` method invocations into lam function invocations and `get` operations into dependency pairs.

If the code of the main is

```
new C()!m1( new C(), new C() )
```

then the evaluation of the program will lock objects as depicted below



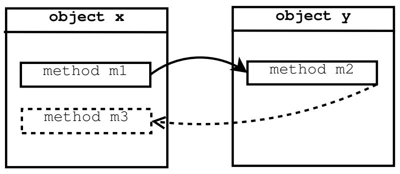
and no deadlock will appear. Since in this case the lam program is not recursive, our analysis technique is straightforward: just evaluate completely the lam program and verify whether the final lam is circularity-free. In fact, we get:

$$\begin{aligned}\Big\langle \mathbb{I}_{x,y,z},\ \mathtt{m1}(x,y,z)\Big\rangle &\longrightarrow \Big\langle \mathbb{I}_{x,y,z},\ (x,y)\|\mathtt{m2}(y,z)\Big\rangle \longrightarrow \Big\langle \mathbb{I}_{x,y,z},\ (x,y)\|(y,z)\|\mathtt{m3}(z)\Big\rangle\\ &\longrightarrow \Big\langle \mathbb{I}_{x,y,z},\ (x,y)\|(y,z)\|\mathtt{0}\Big\rangle\end{aligned}$$

which has no circularity. On the contrary, if the code of `main` is

```
C x = new C(); x!m1( new C(), x )
```

then a deadlock will occur. In fact, in this case $x = z$ and the thread executing `y!m2(x)` will block waiting for the result of `x!m3()`. In turn, this method will never be executed since the object `x` is locked by the initial invocation, as depicted below



This circularity is exactly what our analysis is able to catch. In this case, the main lam is $\texttt{m1}(x,y,x)$ and the computation is

$$\begin{aligned}\Big\langle \mathbb{I}_{x,y},\ \texttt{m1}(x,y,x)\Big\rangle &\longrightarrow \Big\langle \mathbb{I}_{x,y},\ (x,y)\|\texttt{m2}(y,x)\Big\rangle \longrightarrow \Big\langle \mathbb{I}_{x,y},\ (x,y)\|(y,x)\|\texttt{m3}(x)\Big\rangle\\ &\longrightarrow \Big\langle \mathbb{I}_{x,y},\ (x,y)\|(y,x)\|\texttt{0}\Big\rangle\end{aligned}$$

which has a final state with a circularity.

Scheduler's choices and deadlocks. Deadlocks that are usually difficult to detect are those caused by scheduler's choices. Consider the following class `D`

```
class D {
   D m1( D y, D z ) { return new D()!m4(y,z).get ; z!m2(y) ; new D() ; }
   D m2( D y ){ return y!m3( ).get ; }
   D m3( ) { return new D( ) ; }
   D m4( D y, D z ) { return y!m2(z) ; new D() ; }
}
```

and evaluate `(new D)!m1(new D, new D)`. This evaluation is nondeterministic: it may yield a deadlock or not according to the scheduling of the threads. In particular, the execution terminates successfully if the lock of the object `z` is grabbed by `z!m3()` (inside `m2`) before the invocation of `z!m2(y)` (inside `m1`) obtains the lock. If, on the contrary, the lock on `z` is taken by the invocation of `z!m2(y)` inside `m1`, then a deadlock occurs. Let us analyze the program with our technique. The lam functions associated to the above methods are:

$$\begin{aligned}&\texttt{m1}(x,y,z) = (x,u)\|\texttt{m4}(u,y,z)\ ;\ \texttt{m2}(z,y)\|\texttt{m2}(y,z)\ ,\\ &\texttt{m2}(x,y) = (x,y)\|\texttt{m3}(y)\ ,\\ &\texttt{m3}(x) = \texttt{0}\ ,\\ &\texttt{m4}(x,y,z) = \texttt{m2}(y,z)\ ,\end{aligned}$$

In this case, the lam of `m1` has shape $\texttt{T}_1\,\texttt{;}\,\texttt{T}_2$ because the corresponding code spawns two threads in sequence: the invocation of `m2` after the termination of the invocation of `m4`. In turn, since the invocation of `m4` spawns an asynchronous behavior (the asynchronous invocation of `m2`), then $\texttt{T}_2$ also contains an invocation of `m2` caused by `m4`. That is, the leftmost invocation of `m2` in $\texttt{T}_2$ is due to the code of `m1`, the rightmost one is due to the invocation of `m4`.

As before, this lam program is not recursive and, in order to analyze it, we have to compute the final state, assuming $\texttt{m1}(x,y,z)$ as the main function:

$$\begin{aligned}\Big\langle \mathbb{I}_{x,y,z},\ \texttt{m1}(x,y,z)\Big\rangle &\longrightarrow \Big\langle \mathbb{I}_{x,y,z}\oplus(x,y,z<u),\ (x,u)\|\texttt{m4}(u,y,z)\ ;\ \texttt{m2}(z,y)\|\texttt{m2}(y,z)\Big\rangle\\ &\longrightarrow \Big\langle \mathbb{I}_{x,y,z}\oplus(x,y,z<u),\ (x,u)\|\texttt{m2}(y,z)\ ;\ \texttt{m2}(z,y)\|\texttt{m2}(y,z)\Big\rangle\\ \longrightarrow \dots &\longrightarrow \Big\langle \mathbb{I}_{x,y,z}\oplus(x,y,z<u),\ (x,u)\|(y,z)\ ;\ (z,y)\|(y,z)\Big\rangle\end{aligned}$$

thus manifesting a circularity.

The cooperative factorial function. Let us extend the language of Section 2.1 with the primitive type `int`, the arithmetic operations and the conditional expression. The meanings of these features are standard. Then consider the following class `Maths` where the method `fact(n)` computes the factorial of the of the argument (when positive) by performing a recursive invocation on a newly created object. The evaluation produces a finite chain of newly created threads waiting for the termination of the next thread on a new object.

```
class Maths {
   int fact(int n) { return if (n==0) then 1 ;
                             else n*((new Maths)!fact(n-1).get) ; }
}
```

The evaluations of `fact(n)` never yield a deadlock, since no circular dependency can be constructed. This may be verified by translating the program into the lam formalism

$$\big(\ \texttt{fact}(x) = (x,y)\|\texttt{fact}(y),\ \texttt{fact}(x)\big)$$

(the integer parameter is abstracted away, and the method shows only the parameter x representing the object `this`) and evaluating $\texttt{fact}(x)$ till the saturated state, as discussed in Section 3. We notice that the conditional expression is translated into a lam by gathering the dependencies of the two branches, a standard technique in static analysis (in this case, the dependencies of the `then`-branch are empty).

A complex recursive pattern. As an example of recursive program, consider the following `FJf` class:

```
class R { R m1(R y, R z){ return y!m1(z,new R()).get ; z } }
```

The method `m1` is a recursive method (of the same kind of the foregoing factorial function) that invokes `m1` on its first argument with the second argument that is a new object. It does not generate any circularity when invoked with different arguments. However, several concurrent instances of its may produce a circular dependency. For instance, consider the main expression containg *two* invocations of `m1` with the swapped arguments:

```
R x = new R() ; R y = new R() ; R z = new R() ; x!m1(y,z); x!m1(z,y)
```

(we are using *local variables*; these may be easily encoded in our calculus as arguments of additional auxiliary methods). The associated lam program is

$$\big(\ \texttt{m1}(x,y,z) = (x,y)\|\texttt{m1}(y,z,w)\ ,\ \texttt{m1}(x,y,z)\|\texttt{m1}(x,z,y)\ \big)$$

which is linear. Our theory guarantees that, unfolding six times the two invocations of `m1`, it is possible to establish the circularity-freeness of the program. This means that, in order to get the saturated state we have a computation of length twelve. In particular, after four steps of the computation, we get

$$\begin{aligned}\Big\langle \mathbb{I}_{x,y,z},\ \texttt{m1}(x,y,z)\|\texttt{m1}(x,z,y)\Big\rangle &\longrightarrow^2\ \Big\langle \mathbb{V}_1,\ (x,y)\|(x,z)\|\texttt{m1}(y,z,u)\|\texttt{m1}(z,y,v)\Big\rangle\\ &\longrightarrow^2\ \Big\langle \mathbb{V}_2,\ (x,y)\|(x,z)\|(y,z)\|(z,y)\|\texttt{m1}(z,u,u')\|\texttt{m1}(y,v,v')\Big\rangle\end{aligned}$$

where $\mathbb{V}_1 = \mathbb{I}_{x,y,z} \oplus (x, y, z < u) \oplus (x, y, z < v)$ and $\mathbb{V}_2 = \mathbb{V}_1 \oplus (y, z, u < u') \oplus (y, z, v < v')$. Since the last state has a circular dependency, we can stop the analysis here and deduce that the corresponding program deadlocks.

Another complex recursive pattern. The following `FJf` class manifest another issue about saturation. Let

```
class Rec {
    Rec m1(Rec y, Rec z, Rec w){
        return z!m2(y) ; this!m2(z) ; w!m2(z) ; y!m1(this,w,new Rec()) ; z ;
    }
    Rec m2(Rec y){ return y!m3().get; }
    Rec m3() { new Rec(); }
}
```

and let `new Rec()!m1(new Rec(), new Rec(), new Rec())` be the main expression. The lam function associated to the above methods are

$$\begin{aligned}
&\mathtt{m1}(x, y, z, w) = \mathtt{m2}(z, y) \| \mathtt{m2}(x, z) \| \mathtt{m2}(w, z) \| \mathtt{m1}(y, x, w, w') \ ,\\
&\mathtt{m2}(x, y) = (x, y) \| \mathtt{m3}(y) \ ,\\
&\mathtt{m3}(x) = \mathtt{0} \ ,
\end{aligned}$$

and our theory guarantees that the saturated state is obtained after four unfoldings of `m1` (and completely unfolding the auxiliary method invocations `m2` and `m3`). However, we notice that it is possible to obtain a state with a circularity after two unfoldings of $\mathtt{m1}(x, y, z)$ (which we assume as the main function):

$$\begin{aligned}
\left\langle \mathbb{I}_{x,y,z},\ \mathtt{m1}(x, y, z) \right\rangle &\longrightarrow^* \left\langle \mathbb{V},\ (z, y) \| (x, z) \| (w, z) \| \mathtt{m1}(y, x, w) \right\rangle \\
&\longrightarrow^* \left\langle \mathbb{V}',\ (z, y) \| (x, z) \| (w, z) \| (w, x) \| (y, w) \| (w', w) \| \mathtt{m1}(x, y, w') \right\rangle
\end{aligned}$$

where $\mathbb{V} = \mathbb{I}_{x,y,z} \oplus (x, y, z < w)$ and $\mathbb{V}' = \mathbb{V} \oplus (y, x, w < w')$. In particular, the last state has circularity $(z, y) \| (y, w) \| (w, z)$. However, if we perform a further unfolding of $\mathtt{m1}(x, y, w')$ we get the relation

$$(z, y) \| (x, z) \| (w, z) \| (w, x) \| (y, w) \| (w', w) \| (w', y) \| (x, w') \| (w'', w')$$

that manifests the additional circularity (w, x) (x, w') (w', w), which has no counterpart in the previous states (it cannot be mapped to a past circularity). That is, in order to have a complete account of circular dependencies, it is necessary to compute the lam till the saturated state.

The cooperative fibonacci function. A paradigmatic program that yields a non-linear lam is the one computing fibonacci numbers. Consider to augment the above class `Maths` with the following method `fib`

```
int fib(int n) {
  return if n==0 then 1 ;
     else if n==1 then 1 ;
     else new Maths()!fib(n-1).get + new Maths()!fib(n-2).get ; }
}
```

that implements the standard recursive algorithm of fibonacci. As in the factorial example, the above code is a cooperative solution: the recursive invocations are performed on new objects every time, so that the result is computed in parallel threads. The values of the spawned threads are retrieved by `get` operations, which corresponds to synchronization points. To analyze this program, we consider the associated lam:

$$\big(\ \texttt{fib}(x) = (x,y) \| (x,z) \| \texttt{fib}(y) \| \texttt{fib}(z),\ \texttt{fib}(x)\ \big)$$

(as before, the integer parameter is abstracted away and the method only carries the parameter x representing the object `this`). This lam program is not linear and, as discussed in Section 3, it is circularity-free.

5 Additional issues

Two relevant extensions of the language `FJf` are field assignment and the operation `await`. In this section we indicate the techniques we are developing for coping with them.

Fields and assignments. The update operation increases the difficulties of the deadlock analysis. In particular, a field of an object that is modified by concurrent threads has a nondeterminate final value. This, in turn, may cause unpredictable behaviors due to the scheduling of the threads (see also Section 4). In [7] there is a preliminary study of this issue. For instance, consider the two classes `E` and `F` below

```
class E {
      E m1( F y ) { return  y!n1 ; new E() ; }
      E m2( ) { return new E( ); }
 }
class F {
      E f;
      E n1( ) { return  f = this ; }
      E n2( ) { return f!m2().get ; }
 }
```

and the main expression

```
  F x = new F( new E() ) ; new E()!m1(x) ; x!n2() ;
```

(as before, we are using *local variables*). The evaluation of this expression yields a thread (spawned inside `m1`) that modifies the field `f` of `x` (because of the invocation `x!n1`). This thread is concurrent with the invocation `x!n2()`. If the execution flow is such that the invocation `f.m2()` (inside `x!n2()`) is evaluated before the assignment, then the execution will terminate (with a new object stored in `f`), otherwise `f` will contain a reference to `this` and the computation deadlocks.

In order to take into account the updates, we extend lams with the possibility of specifying *sets* of objects, which model the possible values that may be stored in fields. In particular, the lam program associated to the above code is

$$\big(\ \mathtt{m1}(x,y,\mathcal{Y}) = \mathtt{n1}(y,\mathcal{Y}),\ \mathtt{m2}(x) = \mathtt{0},\ \mathtt{n1}(z,\mathcal{Z}) = \mathtt{0},\ \mathtt{n2}(z,\mathcal{Z}) = \mathtt{m2}(\mathcal{Z})\|(z,\mathcal{Z}),\\ \mathtt{m1}(x,y,\{x,y\})\|\mathtt{n2}(y,\{y,z\})\ \big)$$

where $\mathcal{Y}$ is the set of possible values the fields of the object y may be assigned. Analogously for $\mathcal{Z}$ with respect to z. Notice that the first arguments of `m1` and `m2` are of class `E`, therefore they do not have any fields. This is why they do not have any associated set. The body of `n2` has the pair $(z,\mathcal{Z})$, which represent the set of pairs $\{(z,z') \mid z' \in \mathcal{Z}\}$. The execution of the lam program will be the following:

$$\begin{aligned}
\Big\langle \mathbb{I}_{x,y,z},\ \mathtt{m1}(x,y,\{x,y\})\|\mathtt{n2}(y,\{y,z\})\Big\rangle &\longrightarrow \Big\langle \mathbb{I}_{x,y,z},\ \mathtt{n1}(y,\{x,y\})\|\mathtt{n2}(y,\{y,z\})\Big\rangle\\
&\longrightarrow \Big\langle \mathbb{I}_{x,y,z},\ \mathtt{n2}(y,\{y,z\})\Big\rangle\\
&\longrightarrow \Big\langle \mathbb{I}_{x,y,z},\ (y,\{y,z\})\|\mathtt{m2}(\{y,z\})\Big\rangle\\
&\longrightarrow \Big\langle \mathbb{I}_{x,y,z},\ (y,\{y,z\})\Big\rangle \longrightarrow \Big\langle \mathbb{I}_{x,y,z},\ (y,y)\|(y,z)\Big\rangle
\end{aligned}$$

The final state has a circular dependency, i.e. (y,y).

Await and livelocks. A further synchronization operator of ABS is `await` (see [8, 5] for its formal semantics and examples). The operation `await` suspends the current thread, by releasing the lock on its own object, while waiting for a thread termination. Later on, the thread competes again for grabbing the lock and, when acquired, it tries again for the result. If it is available, the computation proceeds, otherwise the lock is released again and so on. With this semantics, a circular dependency does not correspond to a blocking situation, but to a situation where one or more threads are caught in an infinite loop of getting and releasing the lock. This phenomenon is usually called a *livelock*.

In the presence of livelocks, a circular dependency may not necessarily be a bad situation. Let us discuss this issue. A dependency pair (x,y) in a livelock analysis means that *some thread on* x is waiting – not busy-waiting – for the result of *some thread on* y. Under this meaning, the term $(x,y)\|(y,x)$, which is signaled as an incorrect state by our technique, may be safe because the involved threads can be different. In fact, in this case, since the threads do not busy-wait on the objects x and y, the computation terminates successfully.

In order to distinguish between the two types of circularities, we extend the names used in lams with *thread names*. In this extension – lams with two sorts of names, thread names have a "type", which is the (object) name –, the livelocks are those manifested by terms such as $(\mathtt{t},\mathtt{t}')\|(\mathtt{t}',\mathtt{t})$; while terms as $(\mathtt{t},\mathtt{t}')\|(\mathtt{t}',\mathtt{t}'')$, even if $\mathtt{t}$ and $\mathtt{t}''$ have the same object type, are painless.

6 Conclusion

This paper surveys a technique for the static analysis of deadlocks. This technique associates abstract descriptions, called lams, to programs and then evaluates such descriptions to catch circular dependencies. The technique does not use any pre-defined partial order of resources and does account for dynamic resource creation. We stuck to a popularizing exposition; therefore the technical details have been omitted and several examples should help in clarifying the difficult points.

We are currently experiencing our technique in the HATS European project (`www.hats-project.eu`) on large programs written in an object-oriented language with futures [4]. An inference systems for deriving lams [5], for a subset of this language, has been defined and additional annotations allow us to instruct the inference system when structured data types and iterations occur.

Lams have been used in the first place for detecting the so-called *resource allocation deadlocks*, as encountered in e.g. operating systems. However, our technique seems also adequate for deadlocks due to process synchronizations, as those in process calculi [11, 10, 9]. A thorough comparison with these works is scheduled for the next future.

An interesting direction of research is the application of our algorithm for static analysis to verify properties different than deadlocks. This might boil down to devise languages different than lams and to different definitions of saturated states.

References

1. Agarwal, R., Bensalem, S., Farchi, E., Havelund, K., Nir-Buchbinder, Y., Stoller, S.D., Ur, S., Wang, L.: Detection of deadlock potentials in multithreaded programs. IBM Journal of Research and Development 54(5), 3 (2010)
2. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe program.: preventing data races and deadlocks. In: Proc. OOPSLA '02. pp. 211–230. ACM (2002)
3. Comtet, L.: Advanced Combinatorics: The Art of Finite and Infinite Expansions. Dordrecht, Netherlands (1974)
4. Giachino, E., Grazia, C.A., Laneve, C., Lienhardt, M., Wong, P.Y.H.: Deadlock analysis in practice (2013), submitted. Available at `www.cs.unibo.it/~laneve`
5. Giachino, E., Laneve, C.: Deadlock and livelock analysis in concurrent objects with futures (2013), Technical Report. Available at `www.cs.unibo.it/~laneve`
6. Giachino, E., Laneve, C.: Mutations, flashbacks and deadlocks (2013), submitted. Available at `www.cs.unibo.it/~laneve`
7. Giachino, E., Lascu, T.A.: Lock analysis for an asynchronous object calculus. In: ICTCS 2012 (2012)
8. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Proc. of FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer-Verlag (2011)
9. Kobayashi, N.: Type systems for concurrent programs. In: 10th Anniversary Colloquium of UNU/IIST. LNCS, vol. 2757, pp. 439–453. Springer (2003)
10. Kobayashi, N.: A new type system for deadlock-free processes. In: Proc. CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer (2006)
11. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, ii. Inf. and Comput. 100, 41–77 (1992)