May-Happen-in-Parallel Based Deadlock Analysis for Concurrent Objects

Antonio E. Flores-Montoya, Elvira Albert, Samir Genaim

May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects *

Antonio Flores-Montoya[1], Elvira Albert[2], and Samir Genaim[2]

[1] Technische Universität Darmstadt (TUD), Germany
[2] Complutense University of Madrid (UCM), Spain

Abstract. We present a novel deadlock analysis for concurrent objects based on the results inferred by a points-to analysis and a may-happen-in-parallel (MHP) analysis. Similarly to other analysis, we build a *dependency graph* such that the absence of cycles in the graph ensures deadlock freeness. An MHP analysis provides an over-approximation of the pairs of program points that may be running in parallel. The crux of the method is that the analysis integrates the MHP information within the dependency graph in order to discard unfeasible cycles that otherwise would lead to false positives. We argue that our analysis is more precise and/or efficient than previous proposals for deadlock analysis of concurrent objects. As regards accuracy, we are able to handle cases that other analyses have pointed out as challenges. As regards efficiency, the complexity of our deadlock analysis is polynomial.

1 Introduction

The actor-based paradigm [1] on which concurrent objects are based has evolved as a powerful computational model for defining distributed and concurrent systems. In this paradigm, actors are the universal primitives of concurrent computation: in response to a message, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. The underlying concurrency model of actor languages forms the basis of the programming languages Erlang [3] and Scala [9] that have recently gained in popularity, in part due to their support for scalable concurrency. There are also implementations of actor libraries for Java.

Concurrent objects are actors which communicate via *asynchronous* method calls. Each concurrent object is a monitor and allows at most one *active* task to execute within the object. Scheduling among the tasks of an object is cooperative (or non-preemptive) such that a task has to release the object lock explicitly. Each object has an unbounded set of pending tasks. When the lock of an object is free, any task in the set of pending tasks can grab the lock and

* This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, and by the Spanish projects TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and S2009TIC-1465 *PROMETIDOS-CM*.

start to execute. The synchronization between the caller and the callee methods is performed when the result is strictly necessary. So-called *future variables* are used to decouple method invocation and returned value [6]. The access to values of future variables may require *blocking* the object and waiting for the value to be ready. Thus, blocking and non-blocking asynchronous calls coexist in our framework.

In general, deadlock situations are produced when a concurrent program reaches a state in which one or more tasks are waiting for each other termination and none of them can make any progress. In the concurrent objects paradigm, the combination of non-blocking and blocking mechanisms to access futures may give rise to complex deadlock situations and a rigorous formal analysis is required to ensure deadlock freeness. Similarly to other approaches, our analysis is based on constructing a *dependency graph* which, if acyclic, guarantees that the program is deadlock free. The construction of the graph is done by adding three types of edges between tasks and objects: (1) *task-task* dependency: it indicates that a task is waiting to get a future that another task has to calculate, (2) *task-object* dependency: a task is waiting to get its object's lock, (3) *object-task* dependency: a task is waiting for a future while holding the lock of the object and therefore, making the whole object wait. These dependencies capture all possible deadlock situations that might occur in concurrent objects.

In order to construct the dependency graph, we first perform a points-to analysis [13] which identifies the set of objects and tasks created along any execution. Given this information, the construction of the graph is done by a traversal of the program in which we detect the above types of dependencies. However, without further *temporal* information, our dependency graphs would be extremely imprecise. The crux of our analysis is the use of a precise may-happen-in-parallel (MHP) analysis [2]. Essentially, we label the dependency graph with the program points of the synchronization instructions that introduce the dependencies and, thus, that may potentially induce deadlocks. In a post-process, we discard *unfeasible* cycles in which the synchronization instructions involved in the circular dependency may not happen in parallel. We also describe several extensions to the basic framework to: (1) improve the accuracy of programs that create objects (or tasks) inside loops that are challenging for deadlock analysis, (2) handle concurrent object *groups* and (3) allow the use of future variables as fields. We have implemented our analysis and applied it to formally prove deadlock freeness on an industrial case study developed by Fredhopper®.

2 Language

As in the actor-model, the main idea is that control and data are encapsulated within the notion of concurrent object. This section presents the syntax and semantics of the concurrent objects language, which is basically the same as [11, 8, 4]. A *program* consists of a set of classes, each of them can define a set of fields, and a set of methods. The notation $\bar{T}$ is used as a shorthand for $T_1, ...T_n$, and similarly for other names. The set of types includes the classes and the set

of *future* variable types $fut(T)$. Primitive types and pure expressions pu are omitted for simplicity. The abstract syntax of class declarations CL, method declarations M, types T, variables V, and statements s is:

$$CL ::= \textsf{class}\ C\ \{\bar{T}\ \bar{f}; \bar{M}\} \quad M ::= T\ m(\bar{T}\ \bar{x})\{s; \textsf{return}\ p;\} \quad V ::= x \mid \textsf{this}.f$$
$$s ::= s; s \mid x = e \mid V = x \mid \textsf{await}\ V? \mid \textsf{if}\ p\ \textsf{then}\ s\ \textsf{else}\ s \mid \textsf{while}\ p\ \textsf{do}\ s$$
$$e ::= \textsf{new}\ C(\bar{V}) \mid V!m(\bar{V}) \mid V.\textsf{get} \mid pu \quad T ::= C \mid fut(T) \mid Unit$$

Observe that each object encapsulates a *local heap* which is not accessible from outside this object, i.e., fields are always accessed using the **this** object, and any other object can only access such fields through method calls. We assume that every method ends with a return instruction. We also assume that the program includes a method called main without parameters, which does not belong to any class and has no fields, from which the execution will start. The concurrency model is as follows. Each object has a lock that is shared by all the tasks that belong to the object. Data synchronization is by means of future variables: An **await** y? instruction is used to synchronize with the result of executing task $y{=}x!m(\bar{z})$ such that the **await** y? is executed only when the future variable y is available (i.e., the task is finished). In the meantime, the object's lock can be released and some other *pending* task on that object can take it. In contrast, the expression y.**get** blocks the object (no other task of the same object can run) until y is available, i.e., the execution of $m(\bar{z})$ on x is *finished*.

W.l.o.g, we assume that all methods in a program have different names. As notation, we use $body(m)$ for the sequence of instructions defining method m.

2.1 Operational Semantics

A *program state* S is a set $S = \texttt{Ob} \cup \texttt{T}$ where Ob is the set of all created objects and, and T is the set of tasks (including active, pending and finished tasks). The associative and commutative union operator on states is denoted by white-space. An *object* is a term $ob(o, a, lk)$ where o is the object identifier, a is a mapping from the object fields to their values, and lk the identifier of the *active task* that holds the object's lock or $\perp$ if the object's lock is free. Only one task can be *active* (running) in each object and has its *lock*. All other tasks are *pending* to be executed, or *finished* if they terminated and released the lock. A *task* is a term $tsk(t, m, o, l, s)$ where t is a unique task identifier, m is the method name executing in the task, o identifies the object to which the task belongs, l is a mapping from local (possibly future) variables to their values, and s is the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated and the return value v is available. Created objects and tasks never disappear from the state.

The execution of a program starts from the initial state $S_0 = \{obj(0, f, 0)$ $tsk(0, \textsf{main}, 0, l, body(\textsf{main}))\}$ where we have an initial object with identifier 0 executing task 0. f is an empty mapping (since main had no fields), and l maps local reference and future variables to **null** (standard initialization). The execution proceeds from S_0 by applying *non-deterministically* the semantic rules

depicted in Fig. 1 (the execution of sequential instructions is standard and thus omitted). The operational semantics is given in a rewriting-based style where a step is a transition of the form $a\ \underline{\mathrm{b}} \rightarrow \underline{\mathrm{b}'}\ c$ in which: dotted underlining indicates that term b is rewritten to b′; we look up the term a but do not modify it and hence it is not included in the subsequent state; and term c is newly added to the state. Transitions are applied according to the rules as follows.

New_Object: an active task t in object o creates an object o' of type B, its fields are initialized with default values (init_atts) and o' is introduced to the state with a free lock. Observe that as the previous object o is not modified, it is not included in the resulting state. Activate: A non finished task can obtain its object's lock if it is free. Async_Call: A method call creates a new task (the initial state is created by *buildLocals*) with a fresh task identifier t_1 which is associated to the corresponding future variable y in l'. Await1: If the future variable we are awaiting for points to a finished task, the await can be completed. The finished task t_1 is only looked up but it does not disappear from the state as its return value may be needed later on. Await2: Otherwise, the task yields the lock so that any other task of the same object can take it. Return: When **return** is executed, the return value is stored in v so that it can be obtained by the future variables that point to that task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction $\epsilon(v)$). Get: A $x = y.\mathbf{get}$ instruction waits for the future variable but without yielding the lock. Then, it stores the value associated with the future variable y in x.

3 The Notion of Deadlock

In this section, we formalize the notion of deadlock in concurrent objects in the context of our language. Our notion of deadlock is equivalent to the extended deadlock of [4], which corresponds to the classical definition of deadlock by [10]. As in [4], we distinguish two situations which are instrumental to define the notion of deadlock: a *waiting task* which might be waiting to obtain the lock of the object, or that it is waiting to read a future (using **await** or **get**) that other task has to calculate, and a *blocking task* which is waiting to read a future *and* holds the lock, i.e., using **get**. We refer to the object in which a blocking task is executing as *blocked object*. Possible deadlocks will appear as different combinations of the two synchronization primitives in our language, **await** y? and $y.\mathbf{get}$. Detecting deadlocks in a language using such mechanisms is challenging because of potential inconsistencies between synchronization points in separate, yet cooperating, methods, as we show in the following example.

Example 1. Fig. 2 defines several classes (A, B, C, Cl and Sr) with methods that feature typical synchronization patterns. For simplicity, we omit local variables declarations and, at some points, we do not assign the result of $y.\mathbf{get}$ or method calls to any variable. We illustrate the following types of deadlock: **Selflock** (main1). In this case, there is only object a which introduces a selflock. The call

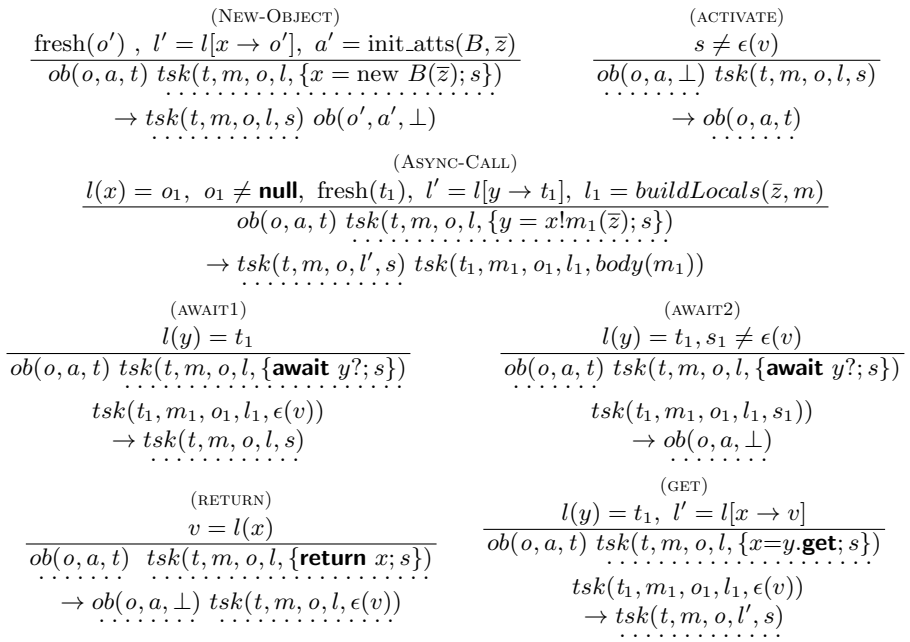


Fig. 1. Summarized semantics of concurrent objects

at L3 to blk1 performs a call to empt on the same object a and gets blocked at L27 waiting for its result. The call to empt on a will never be executed. Here, task blk1 is blocking, task empt is a waiting task, and the object a is blocked. **Mutual lock** (main2). Two objects a and b are created. They both execute task blk1 which makes a call to the other object and waits for the result at L27 without releasing the lock. If both tasks execute in parallel, there will be a deadlock (the two tasks are blocking and the objects are blocked). **Mutual indirect** (main3). Object a starts to execute blk2 and object b will execute blk3. In b, blk3 calls wait on c. Then, b gets blocked until wait on c finishes. Now, wait on c calls empt on a and waits for its termination without holding the lock of c (waiting task). If, in the meantime, blk2 was executing in a and called empt on b blocking object a, then empt will never start to execute as b is blocked in blk3. In summary, objects a, b are blocked, tasks blk2 and blk3 are respectively responsible for blocking the objects, task wait on c and task empt on a are waiting tasks. **MHP** (main4). There is no deadlock in the execution of main4 since it is guaranteed that the execution of acc in L66 will start only after the execution of go at L64 has finished. In particular, when the execution of acc blocks the srv object at L57 waiting for termination of rec it is guaranteed that srv is no longer blocked. We will see that the inference of this information requires the enhancement of the analysis with temporal MHP information.

The following state dependencies are equivalent to the notion of extended deadlock defined in [4] and classical deadlock [10], but adapted to our syntax.

```
1  main1() {
2    a=new A();
3    a!blk1(a);
4  }
5  main2() {
6    a=new A();
7    b=new A();
8    a!blk1(b);
9    b!blk1(a);
10 }
11 main3() {
12   a=new A();
13   b=new B();
14   c=new C();
15   b!blk3(c,a);
16   a!blk2(b);
17 }
18
19 main4() {
20   Sr s=new Sr();
21   s!go();
22 }
23
24 class A {
25   Unit blk1(A a) {
26     f=a!empt();
27     f.get;
28   }
29   Unit blk2(B b) {
30     f=b!empt();
31     f.get;
32   }
33   Unit empt() {}
34 }
35 class B {
36   Unit blk3(C c,A a) {
37     f=c!wait(a);
38     f.get;
39   }
40   Unit empt() {}
41 }
42
43 class C {
44   Unit wait(A a) {
45     f=a!empt();
46     await f?;
47   }
48 }
49
50 class Cl {
51   Sr srv;
52   Unit go(Sr s) {
53     srv=s;
54   }
55 Unit acc() {
56     f=srv!rec("...");
57     f.get;
58   }
59 }
60 class Sr {
61     Unit rec(Str m){}
62     Unit go() {
63      c=new Cl();
64     f=c!go(this);
65     f.get;
66     c!acc();
67   }
68 }
```

Fig. 2. Simple examples featuring different types of deadlock

Definition 1 (state dependencies). *Given a program state $S = \texttt{Ob} \cup \texttt{T}$, we define its dependency graph G_S whose nodes are the existing object and task identifiers and whose edges are defined as follows:*

1. ***Object-Task**: $o \rightarrow t_2$ iff there is an object $obj(o, a, t) \in \texttt{Ob}$ and tasks $tsk(t, m, o, l, \{x = y.\textbf{get}; s\})$, $tsk(t_2, m, o_2, l_2, s_2) \in \texttt{T}$ where $l(y) = t_2$ and $s_2 \neq \epsilon(v)$.*
2. ***Task-Task**: $t_1 \rightarrow t_2$ iff there are two tasks $tsk(t_1, m_1, o_1, l_1, \{sync; s\})$, $tsk(t_2, m_2, o_2, l_2, s_2) \in \texttt{T}$ where $sync \in \{x = y.\textbf{get}, \textbf{await}\ y?\}$, $l_1(y) = t_2$ and $s_2 \neq \epsilon(v)$.*
3. ***Task-Object**: $t \rightarrow o$ iff there is a task $tsk(t, m, o, l, s) \in \texttt{T}$ and an object $obj(o, a, lk) \in \texttt{Ob}$ with $lk \neq t$ and $s \neq \epsilon(v)$.*

The first type of dependency corresponds to the notion of blocking task and blocked object and the other two to waiting tasks. Dependencies are created as long as the task we are waiting for is not finished. Observe that a **get** instruction will generate two dependencies, whereas an **await** will generate only a dependency. Besides, every task without the object's lock (which is not finished) has a dependency to its object.

Example 2. Let us consider the final (deadlock) state for main3 described in Ex. 1. Here, we denote by o:m a task executing method m on object o. We have the following seven dependencies in this state which form a cycle:

d1	a → b:empt	d3	a:empt → a	d5	b → c:wait	d7	c:wait → a:empt
d2	a:blk2 → b:empt	d4	b:blk3 → c:wait	d6	b:empt → b		

Observe that in object a we have a blocking task a:blk2 executing a **get** which induces dependencies d1 and d2 above, and a waiting task a:empt that induces

d3. In b, we have the blocking task b:blk3 that adds d4 and d5 and a waiting task b:empt that adds d6. In c, we have a waiting task c:wait that induces d7.

Definition 2 (deadlock). *A program state S is deadlock iff its dependency graph G_S contains a cycle.*

We are assuming that object fields cannot contain future variables. The removal of this restriction will be discussed in Sec. 5.2. As a consequence, there cannot be cycles involving only task-task dependencies. Intuitively, a cycle that involves only task-task dependencies represents a task that is (possibly indirectly) waiting for itself. Without future fields, a task t can only wait for: other tasks that were created strictly before but did not call t (using futures as parameters); or tasks that were called (possibly indirectly) by t itself. t has no access to its future nor to any of these tasks. Consequently, at least one object must be involved in a cyclic dependency. Additionally, **await** y? instructions only create task-task dependencies. In order to have cycles, we need at least one object-task dependency. Therefore, at least one y.**get** instruction must be involved.

4 Deadlock Analysis

In this section we describe our deadlock analysis which over-approximates the notion of deadlock in Def. 1. If the analysis reports that a program is deadlock-free, then there is no execution that reaches a deadlock state. When the analysis reports a *potential* deadlock, it also provides hints on the program points involved in this deadlock. Our analysis performs two steps: (1) We generate an *abstract* dependency graph $\mathcal{G}$ that over-approximates the dependency graphs G_S of any reachable state S. This graph is obtained by abstracting objects and tasks using points-to analysis. (2) We declare every cycle in $\mathcal{G}$ as a potential deadlock scenario and, in a post-process, we eliminate unfeasible scenarios by discarding those cycles whose involved program points cannot execute in parallel. For the latter, we rely on an MHP analysis. In Sec. 4.1, we describe how points-to analysis is used to abstract objects and tasks. In Sec. 4.2, we present our notion of abstract dependency graph.

4.1 Abstract Tasks and Abstract Objects

Abstracting objects is an extensively studied problem in program analysis. It is at the heart of almost any static analysis for object oriented programs, and usually is referred to as points-to analysis [13]. In principle, any points-to analysis can be used to obtain the information we require. The choice, however, affects the performance and precision of our deadlock analysis. In what follows, we explain how we use *object-sensitive* [13] points-to analysis in order to abstract not only objects, but also tasks. An analysis is object-sensitive if methods are analyzed separately for the different (sets of) objects on which they are invoked. As objects are the concurrency units, object-sensitive points-to analysis naturally suits our setting since tasks are identified with the objects on which they execute.

Following [13, 15], objects are abstracted to syntactic constructions of the form $ob_{ij...pq}$, where all elements in $ij \ldots pq$ are allocation sites (the program points in which objects are created). The abstract object $ob_{ij...pq}$ represents all run-time objects that were created at q when the enclosing instance method was invoked on an object represented by $ob_{ij...p}$, which in turn was created at allocation site p. As notation, we let A be the set of all allocation sites, $\mathcal{P}$ be the set of program points, $\mathcal{V}$ be the set of variables and $pp(s)$ be the program point where statement s is. Given a constant $k \geq 1$, the analysis computes (i) a finite set of abstract object names $\mathcal{O} \subseteq \{ob_\ell \mid \ell \in A \cup A^2 \cup \cdots \cup A^k\}$; and (ii) a partial function $\mathcal{A} : \mathcal{O} \times \mathcal{P} \times \mathcal{V} \mapsto \wp(\mathcal{O} \cup \{\mathbf{null}\})$, where $\mathcal{A}(ob, p, x)$ is the set of abstract objects to which the *reference variable* x might point to, when executing program point p on the abstract object ob. Constant k defines the maximum length of allocation sequences, and it allows controlling the precision of the analysis and ensuring termination. Allocation sequences may have unbounded length and thus it is sometimes necessary to approximate such sequences. This is done by just keeping the k rightmost positions in sequences whose length is greater than k.

We also use the points-to information for task abstraction. Intuitively, we let $\mathcal{T} = \{ob.m \mid ob \in \mathcal{O},\ m \text{ is a method name}\}$ be the set of abstract task identifiers, where $ob.m \in \mathcal{T}$ represents a task that executes the code of method m on the abstract object ob. The points-to analysis is modified to track the values of future variables (which are task identifiers). We distinguish two kinds of task identifiers: normal tasks $tk \in \mathcal{T}$, whose result might be available or not; and ready tasks $tk_r \in \mathcal{T}_r$, whose result is guaranteed to be available (and therefore will not cause any further waiting). Briefly, the significant changes are: (i) the analysis of $y = x!m(\bar{z})$, in which y is assigned the set of abstract task identifiers tk that are induced by the abstract objects to which variable x points-to; and (ii) the analysis of $y.\mathbf{get}$ and $\mathbf{await}\ y?$, in which each $tk \in \mathcal{A}(ob, p, y)$ is substituted by tk_r. In order to use this information, we abuse notation and assume that function $\mathcal{A}$ is extended to map future variables to elements of $\wp(\mathcal{T} \cup \mathcal{T}_r)$. We use function α to denote the mapping from concrete object and task identifiers to corresponding abstract ones.

Example 3. Let us consider the analysis of method main2. The objects created at L6 and L7 are abstracted to ob_6 and ob_7 respectively. Thus, the tasks spawned at L8 and L9 are abstracted to ob_6.blk1 and ob_7.blk1, respectively. Within the two executions of blk1, new tasks executing empt are spawned. They are executed on the object that is passed as parameter. Hence, we keep two separate abstractions, ob_6.empt for the task executing on ob_6, and ob_7.empt for the one executing on ob_7. Next, the future variable f is assigned the abstract value of the tasks whose termination is waiting for. Thus, the value of f is abstracted to ob_7.empt for the task executing on ob_6 and to ob_6.empt for the one executing on ob_7. Note that the use of object-sensitive information is fundamental for precision. Using object-insensitive analysis, all calls to the methods blk1 and empt had been abstracted by the same abstract task identifier (instead of keeping the two identifiers separate), and thus had led to an utter lose of precision.

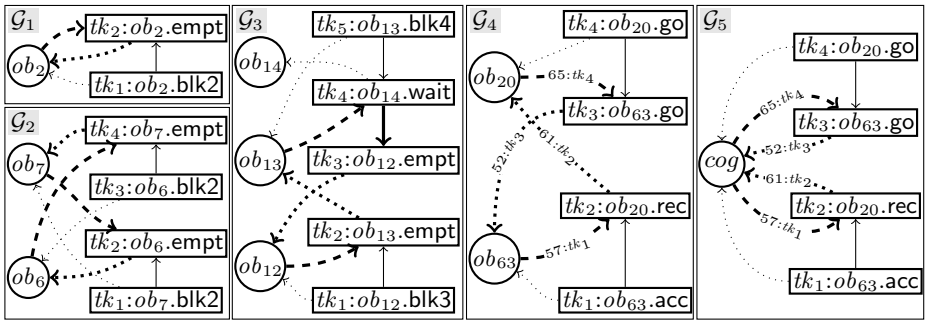


Fig. 3. ($\mathcal{G}_1 - \mathcal{G}_4$) abstract dependency graphs for main1-main4 of Fig. 2; ($\mathcal{G}_5$) Abstract dependency graphs for main4 with COGs;

4.2 Abstract Dependency Graph

Given the object and task abstractions provided in Sec. 4.1, we can construct an *abstract dependency graph* for the program as follows.

Definition 3 (abstract dependency graph). *The abstract dependency graph is a directed graph $\mathcal{G}$, whose nodes are $\mathcal{O} \cup \mathcal{T}$, and whose edges are:*

1. ***Object-Task**: $ob \xrightarrow{p:tk} tk'$ iff there is an instruction $x = y.\textbf{get}$ at program point $p \in \mathcal{P}$, $tk, tk' \in \mathcal{T}$ and $ob \in \mathcal{O}$ such that $tk = ob.m$ and $tk' \in \mathcal{A}(ob, p, y)$.*
2. ***Task-Task**: $tk \xrightarrow{p:tk} tk'$ iff there is an instruction $x = y.\textbf{get}$ or $\textbf{await}\ y?$ at program point $p \in \mathcal{P}$, $tk, tk' \in \mathcal{T}$ such that $tk = ob.m$ and $tk' \in \mathcal{A}(ob, p, y)$.*
3. ***Task-Object**: $tk \xrightarrow{p:tk} ob$ iff $tk \in \mathcal{T}$ and $ob \in \mathcal{O}$ such that $tk = ob.m$, where p is the entry program point of m, or of an instruction $\textbf{await}\ y?$ in m.*

Observe that the construction follows exactly Def. 1, but we use abstract information instead of the concrete one. The nodes are the abstract objects and tasks, and the edges represent the following information: (1) the abstract object ob is locked by the abstract task tk until task tk' finishes; (2) the abstract task tk is waiting for the abstract task tk' to finish; and (3) the abstract task tk might be waiting on the abstract object ob. The labels on the edges $p{:}tk$ keep information on the source of this edge, p is a program point in the task tk. These labels will be used below. Roughly, the abstract dependency graph can be seen as an abstraction of the graph that results from the union of all G_S, where some nodes are collapsed. Note that ready tasks tk_r are ignored as they cannot involve any waiting.

Example 4. Fig. 3 shows the abstract dependency graphs obtained from the analysis of the four main methods in Fig. 2 ($\mathcal{G}_5$ will be explained later). Cycles are marked with bold edges. The deadlocks informally described in Ex. 1 for the first three main methods can be seen in the graphs. We have omitted the labels in

these three graphs as they are not relevant. For instance, in $\mathcal{G}_3$, the cycle includes the two blocked objects a (here ob_{12}) and b (here ob_{13}) and the three waiting tasks as described in Ex. 1. An important point to note is that $\mathcal{G}_4$ contains a cycle. However, as justified informally in Ex. 1, the program is deadlock free. The problem is that the graph does not contain *temporal* information about whether the instructions involved in the cycle may indeed happen in parallel.

The MHP analysis of [2] is adapted to infer a set of symmetric pairs $\mathcal{M} \subseteq ((\mathcal{P} \times \mathcal{T}) \times (\mathcal{P} \times \mathcal{T}))$ with the following guarantee: for any reachable state S, if $tsk(t_1, m_1, o_1, l_1, s_1)$ and $tsk(t_2, m_2, o_2, l_2, s_2)$ are two tasks of S available in the state such that $t_1 \neq t_2$, then $(p_1{:}tk_1, p_2{:}tk_2) \in \mathcal{M}$ where $p_i = pp(s_i)$ and $tk_i = \alpha(t_i)$. Intuitively, if program points p_1 and p_2 might execute in parallel within tasks t_1 and t_2, then $\mathcal{M}$ includes this information at the level of the corresponding abstract tasks.

Example 5. As an example, the application of the MHP [2] to main2 gives, among others, the MHP pair $(27{:}tk_3, 27{:}tk_4)$ where tk_3 and tk_4 are shortcuts given in Fig. 3 for ob_6:blk1 and ob_7:blk1, respectively. This pair indicates that objects ob_6 and ob_6 can be executing the **get** instruction at program point 27 in parallel. Thus, a deadlock would occur. The MHP of main4 gives the following set of MHP pairs $\{(61{:}tk_2, 57{:}tk_1), (65{:}tk_4,\ 52{:}tk_3)\}$. The important point to notice is that instructions $65{:}tk_4$ and $57{:}tk_1$ cannot happen in parallel. This formally justifies the intuition for deadlock freeness given in Ex. 1.

Definition 4 (feasible cycle). *A cycle* $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1 \in \mathcal{G}$ *is feasible iff* $(p_i{:}tk_i, p_j{:}tk_j) \in \mathcal{M}$ *for all* $1 \leq i < j \leq n$*, and at least one* e_i *is an object identifier.*

As we have mentioned before, a deadlock must involve at least one object which is blocked. Also, all points that are involved in the considered cycle must happen in parallel (this is the condition over-approximated by MHP). In the implementation, instead of first building the dependency graph and then checking the feasibility of cycles, for each cycle, there is an interleaved construction such that new dependencies are only added if they satisfy the MHP condition with respect to the previous ones.

Example 6. The cycle in $\mathcal{G}_5$ is not feasible because $(65 : tk_4, 57 : tk_1)$ does not belong to the MHP pairs given in Ex. 5.

Our soundness theorem ensures that if there is a deadlock in the execution of the concrete program, then the abstract graph contains a feasible cycle.

Theorem 1 (soundness). *Let* S *be a reachable state. If there is a cycle* $e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_1$ *in* G_S*, then* $\alpha(e_1) \xrightarrow{p_1:tk_1} \alpha(e_2) \xrightarrow{p_2:tk_2} \cdots \xrightarrow{p_n:tk_n} \alpha(e_1)$ *is a feasible cycle of* $\mathcal{G}$*.*

The following corollary follows trivially from the above theorem.

Corollary 1 (deadlock-freeness). *If there are no feasible cycles in* $\mathcal{G}$*, then the program is deadlock-free.*

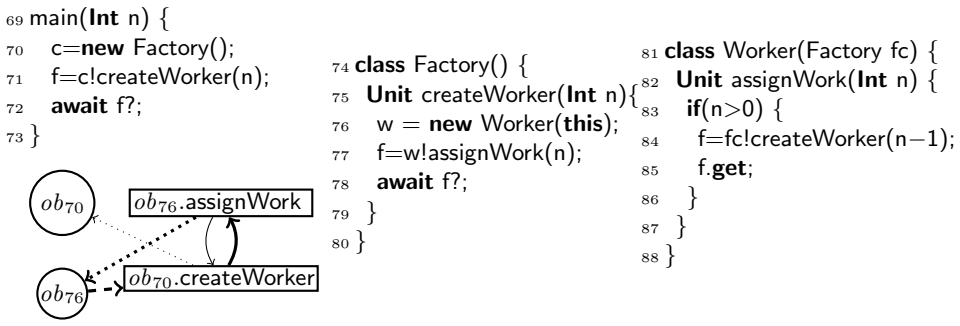


Fig. 4. Challenging example of [8] for handling objects created in loops

5 Extensions of the Basic Framework

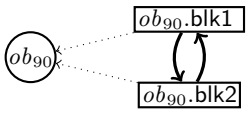
In this section, we present several extensions of our analysis in order to improve the precision on some challenging examples, as well as extensions to handle advanced features such storing future variables in fields, and objects groups.

5.1 Creating Objects and Tasks Inside Loops

Programs that create objects (or tasks) inside loops are challenging for deadlock analysis due to the fact that the created objects, whose amount usually depends on an unknown input value, are abstracted to a finite set of abstract objects (or tasks). This makes it difficult to distinguish their activities and leads to spurious scenarios. The example in Fig. 4 is a program reported in [8] as challenging (they failed prove it deadlock free). When calling main(n), the program creates n objects of type Worker using a single Factory object. The first call createWorker(n) creates one object of type Worker, calls its assignWork method, and waits until this call finishes. Method assignWork in turn calls createWorker(n−1) in order to create the remaining n−1 objects, and waits (blocking the current Worker object) until this call finishes. This program is deadlock free, because every Worker object waits (transitively) for a task that is running on a *different* Worker object. However, our basic framework as described in Sec. 4 generates the abstract dependency graph shown in the figure, which contains a cycle. Node ob_{76} represents all objects of type Worker, and ob_{76}.assignWork represents all tasks of method assignWork.

The cycle represents a scenario in which an object of type Worker (the source object) is blocked waiting (transitively) for another task running on an object of type Worker (the target object) to finish. Since both the source and target objects are represented by the same abstract value ob_{76}, we have to assume the case in which they are equal, and thus create a deadlock. This, however, is a spurious scenario that cannot actually happen. Our aim is to prove that this cycle is unfeasible, in particular that the source and target objects cannot be the same even if they are assigned the same abstract value. Consider the dependency ob_{70}.createWorker $\rightarrow$ ob_{76}.assignWork, and observe that whenever createWorker calls

```
89 main() {
90   A a = new A();
91   a!run();
92 }
```




```
93 class A {
94   Fut<Unit> f;
95   Bool ready=False;
96   Unit run(){
97     ff=this!blk1();
98     f=this!blk2(ff);
99     ready=True;
100  }
```

```
101  Unit blk1(){
102    await ready==True;
103    await f?;
104  }
105  Unit blk2(Fut<Unit> ff){
106    await ff?;
107  }
108 }
```

Fig. 5. Deadlock Analysis for of Future Variables as Fields

assignWork it uses a *fresh* object, i.e., an object that has been created in its local scope. Thus, it cannot be the case that assignWork belongs to the same object that is blocked waiting for createWorker to finish. This simple information is enough to discard this cycle. Our extension is based on identifying edges with this *freshness* property, and then using them to discard cycles as the one above, and several other cases. The appealing feature of the *freshness* property is that it can inferred with almost no further overhead. Briefly, it is done by modifying the points-to analysis to annotate abstract objects with a *freshness flag*, which indicates that it represents a single object that has been recently created in the current scope. Once objects *escape* from their local scope the corresponding flags are discarded.

5.2 Future Variables as Fields

So far we have assumed that future variables are passed (by-value) between methods only as parameters or return values. This restriction guarantees that any deadlock must involve at least one object that is locked by one of its tasks (using instruction y.**get**). In this section, we allow future variables to be defined as fields. This is challenging because a task gains access to its identifier. Consider the program in Fig. 5. It is easy to verify that, starting from method main, we can reach a state in which tasks ob_{90}.blk1 and ob_{90}.blk2 are waiting for each other: ob_{90}.blk2 receives the task identifier of ob_{90}.blk1 as parameter, while ob_{90}.blk1 reads that of ob_{90}.blk2 from a field. By applying our analysis, we get the graph shown in Fig. 5, which does not have any cycle that goes through an object and hence, according to Def. 4, we incorrectly conclude that it is deadlock free. The reason is that we cannot ignore cycles that involve only task-task nodes. Thus, Def. 4 should be modified to drop the requirement that the cycle includes at least one object node.

This change only affects the latter part of the analysis where we explore the abstract dependency graph and the rest of the analysis is still valid. However, we can expect an increment on the number of false positives. Many recursive methods will induce new cycles such as the example in Sec. 5.1: we have to consider the cycle that only involves assignWork and createWorker. Fortunately, we can apply the notion of freshness described in Sec. 5.1 to tasks rather than to

objects and discard most of these new false positives. Briefly, the *task-freshness* property can be used to prove that a task always waits for other (fresh) tasks that have been created in its local scope, and thus it cannot be the case that a task is transitively waiting for itself to finish.

5.3 Object Groups

Object *groups* extend the concurrent objects model to allow grouping objects into concurrency units such that at most one task can be executing among all objects in each group. At the programming language level, this feature is supported by the instruction **new cog** $C(\bar{x})$ which creates a new object of type C and, in addition, assigns it to a *new* group that includes this single object. The instruction **new** $C(\bar{x})$, in turn, creates a new object of type C and assigns it to the group of its parent object (i.e, the object that executed the instruction). The definition of deadlock in this setting is very similar to Defs. 1 and 2. The difference is that object nodes are replaced by group nodes, and the ***Object-Task*** and ***Task-Object*** edges connect a task t with the node that corresponds to the group of o. Then, a program deadlocks iff it reaches a state whose dependency graph includes a cycle with at least one group node. Similarly, at the abstract level, the deadlock property can be approximated as in Sec. 4, by replacing abstract object nodes by abstract group nodes. This, however, requires inferring a set of abstract groups and relating them to the set of abstract objects. In practice, we infer this information by modifying the points-to analysis to track the (abstract) group of each abstract object in a straightforward way. This modification has negligible overhead.

Example 7. Assuming a concurrent object groups setting, method main5 is not deadlock free any more. Our analysis infers that objects created at L20 and L63 belong to the same group. Then, it constructs the abstract dependency graph $\mathcal{G}_6$ depicted in Fig. 3. Note that $\mathcal{G}_6$ merges the two object nodes of $\mathcal{G}_5$ into a single group node. Since $\mathcal{G}_6$ contains a cycle $cog \rightarrow ob_{63} \rightarrow cog$, and moreover $65{:}tk_4$ and $52{:}tk_3$ may execute in parallel, it reports a deadlock. By replacing **new** by **new cog** the code would be deadlock free and detected as such by the analysis.

6 Experiments

We report on DECO[3], a DEadlock analyzer for Concurrent Objects, which implements the analysis described in the paper and the extensions for the complete ABS language [11]. Given a program with a main procedure, the output of the analysis is a description of the potential cycles (if any) so that the user can easily find the causes of the deadlocks and discard false positives. This section aims at experimentally evaluating the accuracy and performance of the DECO tool, and comparing our results with those obtained by the SDA tool [7]. However, such comparison is not always feasible because SDA does not handle complete ABS.

[3] DECO can be tried online at http://costa.ls.fi.upm.es/costabs/deco

		SDA Tool		DECO Tool				
Medium codes	Lines	Result	T(sec)	Result	PtT(ms)	MhpT(ms)	DT(ms)	T(sec)
Bookshop	418	×	-	✓	< 20	< 20	< 20	1.36
PeerToPeer	185	×	-	✓	99	359	< 20	1.63
LeaderElection	63	×	-	✓	< 20	< 20	< 20	1.48
PingPong	66	×	-	✓	< 20	< 20	< 20	1.30
MultiPingPong	89	×	-	18	< 20	< 20	< 20	1.43
BoundedBuffer	103	✓	1.65	✓	< 20	< 20	< 20	1.26
Case Studies	Lines	Result	T(sec)	Result	PtT(ms)	MhpT(ms)	DT(ms)	T(sec)
TradingSystem	1466	×	-	✓	79	491	< 20	3.15
FredHopper	2111	×	-	✓	372	2456	351	6.62
Adpt Fredhopper	2081	✓	65.30	✓	136	1347	73	4.38

Fig. 6. Medium examples and case studies results

Basically, they have to adapt the programs to remove recursive object structures. Otherwise, SDA fails to obtain any result. They also annotate **await** instructions with boolean conditions to increase the precision. Await boolean conditions are instructions of the form **await** `e` where `e` is a boolean expression. This instruction releases the object's lock when the condition is not satisfied. This kind of instructions can be used for internal synchronization within an object. Our analysis supports recursive object structures naturally and we have developed an improvement over the MHP analysis of [2] to take simple **await** instructions with boolean conditions (without function calls) into account. More complex boolean awaits are ignored without harming the soundness of the analysis.

Both tools have been tested on: 39 small examples (19 are taken from [7] and 20 are developed by us); on 6 medium-size programs written for benchmarking purposes by ABS programmers; and two case studies developed by Fredhopper®. The source code of all examples can be found in the above website. All examples were run with a constant $k = 2$ for the poinsto analysis. About the small examples, it is worth mentioning that our tool does not report any false positive, while SDA gives 8 false positives (5 of them are on our examples and 3 are on theirs) and fails to analyze 2 examples. The failures are due to the recursive object structures limitation above. Out of the 8 false positives, one is due to their treatment of loops over data structures and the remaining ones are correctly discarded in our tool thanks to the extension described in Sec. 5.1.

Fig. 6 reports the results on the medium-size benchmarks and on the case studies. The first column contains the name of the benchmarks and the second one its size (number of lines). The leftmost set of columns are the results computed by the SDA tool and the rightmost set by our implementation. In both sets, the first column shows the result of the analysis: × means that the tool fails to analyze it, ✓means that it proves deadlock freeness, and a positive number meaning the number of cycles found. The next column shows the analysis time in seconds (average of 10 runs). For DECO, we show first the analysis time of the different phases of the analysis: (PtT) is points-to analysis time, (MhpT) is MHP time, (DT) is the time spent on creating and exploring the abstract dependency

graph. The rightmost column is the overall time. It can be seen that we have proved deadlock freeness in all medium-size programs except for *MultiPingPong* which reports 18 potential deadlocks. By executing the program step by step we have checked that these potential deadlocks are indeed real deadlocks. SDA was able to analyze only the *BoundedBuffer* example. Analyzing the remaining examples would require rewriting them to avoid recursive object structures.

As regards the case studies, the first two ones are the original versions while the last one is a modification of the second one to avoid the limitations of SDA described above. Our tool can successfully analyze both the original and the modified versions reporting deadlock freeness. As regards performance, all analyses have been performed in an Ubuntu 12.04 64-bit with Intel core i7-3667U 2.00GHz x 4 and 8GiB of Memory. The total times for the two tools have been externally measured, while partial times of our analysis have been measured internally. These times do not take compilation and program initialization into account and thus they do not add to the total time. They also have a limited precision and thus negligible times are often reported as 0. We present them as < 20 in the table. We can see that, for *BoundedBuffer*, both tools have a similar performance (1.65 secs and 1.26 secs). However, when it comes to analyze bigger programs our approach is much more efficient (4.38 secs over 65.30 secs) which seems to indicate that our techniques are more scalable.

7 Related Work and Conclusions

We have presented a novel deadlock analysis for languages with actor-based concurrency based on a points-to analysis and an MHP analysis. We argue that our technique outperforms previous proposals [8, 5, 4, 7] in precision or efficiency and it considers a more expressive language. Experiments suggest we achieve better precision than [8, 7] because our pointsto analysis keeps a more fine-grained representation of objects and their dependencies than their contract-based approach. That, together with Sec. 5.1, gives precise results in examples pointed out as challenging in [8, 7]. Besides, our analysis does not suffer from any of the restrictions mentioned in Sec. 6. The language used in [5] is very restrictive (e.g., it does not have an object creation instruction, nor synchronizations using await, among other limitations). More recent work [4] improves the previous one, since the use of Petri nets specifies the temporal behavior of the methods. However, the language is again more restrictive than ours since it does not allow dealing with objects stored in fields and does not treat while loops and object creation explicitly. Even more importantly, our analysis is polynomial, while [4] requires solving a reachability problem in Petri nets (which is EXPSPACE-hard). Note that the complexity of the points-to analysis can vary depending on the precision, it is almost linear in [16] and [13] has cubic worst-case complexity.

It is not the first time that an MHP analysis has been used to detect deadlocks. Its use dates back to 1991 [12] where it was applied to detect deadlocks on Ada programs. It has been also applied in [14] to thread-based programs. There are some fundamental differences between [14] and our work due also in part

to the differences between the underlying concurrency models. Their algorithm detects locks due to lock-based synchronization whereas wait-notify in Java is not covered. In contrast, we treat wait-notify synchronization and in particular our treatment of future variables (which represent the notify) is very powerful. In their case, the algorithm detects deadlocks between two threads, while our technique does not have this restriction and can detect deadlocks that involve any number of objects. On the other hand, they propose some (unsound) treatment to non-guarded and non-reentrant conditions that target Java programs but cannot happen in our framework.

References

1. G.A. Agha. *A*ctors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, 1986.
2. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *P*roc. of FORTE'12, volume 7273 of *L*NCS, pages 35–51. Springer, 2012.
3. J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *C*oncurrent Programming in Erlang. Prentice Hall, 1996.
4. F. S. de Boer, M. Bravetti, I. Grabe, M. David Lee, M. Steffen, and G. Zavattaro. A Petri Net based Analysis of Deadlocks for Active Objects and Futures. In *P*roc. of FACS 2012, 2012.
5. F. S. de Boer, I. Grabe, and M. Steffen. Termination detection for active objects. *J.* Log. Algebr. Program., 81(4):541–557, 2012.
6. Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In Rocco de Nicola, editor, *P*roc. 16th European Symposium on Programming (ESOP'07), volume 4421 of *L*NCS, pages 316–330. Springer, 2007.
7. E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. Deadlock Analysis of Concurrent Objects – Theory and Practice, 2013.
8. E. Giachino and C. Laneve. Analysis of Deadlocks in Object Groups. In *P*roc. of FMOODS/FORTE, volume 6722 of *L*NCS, pages 168–182. Springer, 2011.
9. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *T*heor. Comput. Sci., 410(2-3):202–220, 2009.
10. R. C. Holt. Some Deadlock Properties of Computer Systems. *A*CM Comput. Surv., 4(3):179–196, 1972.
11. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *P*roc. of FMCO'10 (Revised Papers), volume 6957 of *L*NCS, pages 142–164. Springer, 2012.
12. S. P. Masticola and B. G. Ryder. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *P*arallel and Distributed Debugging, pages 97–107. ACM, 1991.
13. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *A*CM Trans. Softw. Eng. Methodol., 14:1–41, 2005.
14. M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *P*roc. of ICSE, pages 386–396. IEEE, 2009.
15. Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *P*OPL, pages 17–30. ACM, 2011.
16. Bjarne Steensgaard. Points-to analysis in almost linear time. In *S*ymposium on Principles of Programming Languages, pages 32–41, 1996.