Deadlock Analysis of Wait-Notify Coordination

Cosimo Laneve[1⋆][0000−0002−0052−4061] and Luca Padovani[2][0000−0001−9097−1297]

[1] Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus
[2] Dipartimento di Informatica, Università di Torino

Abstract. Deadlock analysis of concurrent programs that contain coordination primitives (`wait`, `notify` and `notifyAll`) is notoriously challenging. Not only these primitives affect the scheduling of processes, but also notifications unmatched by a corresponding `wait` are silently lost. We design a behavioral type system for a core calculus featuring shared objects and `Java`-like coordination primitives. The type system is based on a simple language of object protocols – called *usages* – to determine whether objects are used *reliably*, so as to guarantee deadlock freedom.

1 Introduction

Locks and condition variables [10] are established mechanisms for process coordination that are found, in one form or another, in most programming languages. `Java` provides `synchronized` blocks for enforcing exclusive access to shared objects and `wait`, `notify` and `notifyAll` primitives to coordinate the threads using them: a thread performing a `wait` operation on an object releases the lock on that object and is suspended; a `notify` operation performed on an object awakens a thread suspended on it, if there is one; `notifyAll` is similar to `notify`, except that it awakens all suspended threads. Writing correct concurrent programs using these primitives is notoriously difficult. For example, a thread may block indefinitely if it attempts to lock an object that is permanently owned by a different thread or if it suspends waiting for a notification that is never sent.

We see an instance of non-trivial concurrent `Java` program in Figure 1, which models a coordination problem whereby a single *consumer* retrieves items from two *producers*. Each producer repeatedly generates and stores a new item into a buffer (line 5), notifies the consumer that the item is available (line 6) and waits until the consumer has received the item (line 7). At each iteration, the consumer waits for an item from each buffer (lines 12 and 15) and notifies the corresponding producer that the item has been processed (lines 14 and 17). The main thread of the program forks the producers (lines 25–26) and then runs as the consumer (line 27). In this example, the fact that producers and consumer use the two buffers in mutual exclusion is guaranteed by the syntactic structure of the code, since all accesses to `x` and `y` occur within `synchronized` blocks. However, understanding whether the three threads coordinate correctly so as to realize the desired continuous flow of information is not as obvious. This difficulty is largely

⋆ Research partly supported by the H2020-MSCA-RISE project ID 778233 (BEHAPI).

```
1   public static void producer(Buffer x) {
2       int item = 0;
3       while (true)
4           synchronized (x) {
5               x.Put(item++);
6               x.notify();
7               x.wait();
8           }
9   }
10  public static void consumer(Buffer x, Buffer y) {
11      while (true) {
12          x.wait();
13          System.out.println(x.Get());
14          x.notify();
15          y.wait();
16          System.out.println(y.Get());
17          y.notify();
18      }
19  }
20  public static void main(String[] args) {
21      Buffer x = new Buffer();
22      Buffer y = new Buffer();
23      synchronized (x) {
24          synchronized (y) {
25              new Thread(() -> producer(x)).start();
26              new Thread(() -> producer(y)).start();
27              consumer(x, y);
28          }
29      }
30  }
```

Fig. 1. Multiple-producers/single-consumer coordination in `Java`.

due to the *ephemeral* nature of notifications: a notification sent to a shared object has an effect only if there is another thread waiting to be notified on that object. Otherwise, the notification is lost, with likely undesired implications. For example, suppose to change the program in Figure 1 so that the `synchronized` blocks now in the `main` method (lines 23–24) are moved into the `consumer` method, each protecting accesses to the corresponding buffer. This change would give `producer` and `consumer` a visually appealing symmetric structure, but the correctness of the program would be fatally compromised. Now a producer could lock `x` before the consumer and notify `x` at a time when the consumer is not yet waiting for a notification. Eventually, the consumer would block waiting for a notification on `x` that will never arrive, leading to a deadlock.

The contribution of this paper is a behavioral type system ensuring that well-typed programs using shared objects and coordination primitives are deadlock

free. The type system combines two features inspired by previous works on the static analysis of concurrent programs. First, we use a formulation of the typing rule for parallel compositions akin to that found in linear logic interpretations of session types [18, 3]. Unlike these works, where session endpoints are linear resources, here we apply the typing rule in a setting with shared (hence, non-linear) objects. Second, we rely on behavioral types – called *usages* – to make sure that objects are used *reliably*, ruling out deadlocks due to missing notifications. Kobayashi [12] has already studied a type system based on usages for the deadlock analysis of pi-calculus processes. He shows how to reduce usage reliability to a reachability problem in a Petri net. As is, this reduction does not apply to our setting because the encoding of usages with coordination primitives requires the use of Petri nets with *inhibitor arcs* [2].

The rest of the paper is structured as follows. Section 2 presents a core calculus of concurrent programs featuring threads, shared objects and coordination primitives. Section 3 defines types, usages and the key notion of *reliability*. Typing rules and properties of well-typed programs are given in Section 4. Section 5 discusses related work in more detail and Section 6 concludes. Proofs and the handling of the `notifyAll` primitive can be found in the full paper [15].

2 Language Syntax and Semantics

We define a core language of concurrent programs featuring threads, shared objects and a minimal set of coordination primitives inspired to those of `Java`. We formalize our language as a process calculus comprising standard constructs (termination, conditional behavior, object creation, parallel composition, recursion) in which actions represent coordination primitives on objects. Instead of providing a `synchronized` construct to enforce mutually exclusive access to a shared object, we use explicit acquire and release operations on the object.

Formally, our calculus makes use of a countable set of *variables and object names*, ranged over by x, y, z, and a set of *procedure names*, ranged over by A. A *program* is a pair $(\mathcal{D}, P)$, where $\mathcal{D}$ is a *finite set* of *procedure definitions* of the form $A(\overline{x}) = P_A$, with $\overline{x}$ and P_A respectively being the *formal parameters* and the *body* of A. In a program $(\mathcal{D}, P)$ we say that P is the *main process*. Hereafter we write $\overline{\alpha}$ for possibly empty, finite sequences $\alpha_1, \ldots, \alpha_n$ of various entities. The syntax of processes, expressions and actions is given in Table 1.

Expressions comprise integer constants, variables and object names, and an unspecified set of operators op such as $+$, $\leq$, and so forth. Expressions are evaluated by means of a total function $[\![\cdot]\!]$ such that $[\![x]\!] = x$.

The process done performs no action. The process $\pi.P$ performs the action π and continues as P. The conditional process $\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$ behaves as P if $[\![e]\!] \neq 0$ or else as Q. The process $\mathsf{new}\ x\ \mathsf{in}\ P$ creates a new object x with scope P. The process $\mathsf{fork}\{P\}Q$ forks P and continues as Q. Finally, $A(\overline{e})$ denotes the invocation of the process corresponding to A with actual parameters $\overline{e}$.

An action is either $\mathsf{acq}(x)$, which acquires the lock on x, or $\mathsf{rel}(x)$, which releases the lock on x, or $\mathsf{wait}(x)$, meaning that the process unlocks x and suspends

Table 1. Syntax of the language with runtime syntax marked by †.

Expression	e	::=	n	(constant)
		|	x	(variable)
		|	$e\ \mathsf{op}\ e$	(operator)
Process	P, Q	::=	done	(termination)
		|	$\pi.P$	(action prefix)
		|	$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	(conditional)
		|	$\mathsf{new}\ x\ \mathsf{in}\ P$	(new object)
		|	$\mathsf{fork}\{P\}Q$	(new process)
		|	$A(\overline{e})$	(invocation)
Action	π	::=	$\mathsf{acq}(x)$	(acquire)
		|	$\mathsf{rel}(x)$	(release)
		|	$\mathsf{wait}(x)$	(wait)
		|	$\mathsf{wait}(x, n)$	(waiting)†
		|	$\mathsf{notify}(x)$	(notify)

waiting for a notification, or $\mathsf{notify}(x)$ that notifies a process waiting on x, if there is any. Objects are reentrant and can be locked multiple times by the same process. The prefix $\mathsf{wait}(x, n)$ is a runtime version of $\mathsf{wait}(x)$ that keeps track of the number of times (n) x has been locked before $\mathsf{wait}(x)$ was performed. User programs are not supposed to contain $\mathsf{wait}(x, n)$ prefixes. We write $\mathsf{acq}(x)^n.P$ in place of $\mathsf{acq}(x) \cdots \mathsf{acq}(x).P$ where there are n subsequent acquisitions of x.

To define the operational semantics of a program we make use of an infinite set of *process identifiers*, ranged over by t and s, and *states*, which are pairs of the form $\mathcal{H} \Vdash \mathcal{P}$ made of a *heap* $\mathcal{H}$ and a *process pool* $\mathcal{P}$. Heaps are finite maps from object names to pairs of the form t, n where t identifies the process that has locked an object x and n is the number of times x has been acquired. We allow t to be the distinguished name $\bullet$ and n to be 0 when x is unlocked. Process pools are finite maps from process identifiers to processes and represent the set of processes running at a given time. In the following we occasionally write $\mathcal{H}$ as $\{x_i : t_i, n_i\}_{i \in I}$ and $\mathcal{P}$ as $\{t_i : P_i\}_{i \in I}$. We also write $\mathcal{H}, \mathcal{H}'$ for the union of $\mathcal{H}$ and $\mathcal{H}'$ when $dom(\mathcal{H}) \cap dom(\mathcal{H}') = \emptyset$. Similarly for $\mathcal{P}, \mathcal{P}'$.

The operational semantics of a program $(\mathcal{D}, P)$ is determined by the transition relation $\longrightarrow_{\mathcal{D}}$ defined in Table 2 applied to the initial state $\emptyset \Vdash \mathit{main} : P$. To reduce clutter, for each rule we only show those parts of the heap and of the process pool that are affected by the rule. For example, the verbose version of [R-DONE] is $\mathcal{H} \Vdash t : \mathsf{done}, \mathcal{P} \longrightarrow_{\mathcal{D}} \mathcal{H} \Vdash \mathcal{P}$. Rules [R-ACQ-*] and [R-REL-*] model the acquisition and release of a lock. There are two versions of each rule to account for the fact that locks are reentrant and can be acquired multiple times. Rule [R-WAIT] models a process that unlocks an object x and suspends waiting for a notification on it. The number n of acquisitions is stored in the runtime prefix $\mathsf{wait}(x, n)$ so that, by the time the process is awoken by a notification, it re-acquires x the appropriate number of times. Rules [R-NFY-*] model notifications. The two rules

Table 2. Reduction rules.

$$
\begin{array}{lrcll}
\text{[R-ACQ-1]} & x : \bullet, 0 \Vdash t : \mathsf{acq}(x).P & \longrightarrow_{\mathcal{D}} & x : t, 1 \Vdash t : P & \\
\text{[R-ACQ-2]} & x : t, n \Vdash t : \mathsf{acq}(x).P & \longrightarrow_{\mathcal{D}} & x : t, n+1 \Vdash t : P & \text{if } n > 0 \\
\text{[R-REL-1]} & x : t, 1 \Vdash t : \mathsf{rel}(x).P & \longrightarrow_{\mathcal{D}} & x : \bullet, 0 \Vdash t : P & \\
\text{[R-REL-2]} & x : t, n+1 \Vdash t : \mathsf{rel}(x).P & \longrightarrow_{\mathcal{D}} & x : t, n \Vdash t : P & \text{if } n > 0 \\
\text{[R-WAIT]} & x : t, n \Vdash t : \mathsf{wait}(x).P & \longrightarrow_{\mathcal{D}} & x : \bullet, 0 \Vdash t : \mathsf{wait}(x, n).P & \text{if } n > 0 \\
\text{[R-NFY-1]} & \Vdash t : \mathsf{wait}(x, n).P, s : \mathsf{notify}(x).Q & \longrightarrow_{\mathcal{D}} & \Vdash t : \mathsf{acq}(x)^n.P, s : Q & \\
\text{[R-NFY-2]} & \Vdash t : \mathsf{notify}(x).P, \mathcal{P} & \longrightarrow_{\mathcal{D}} & \Vdash t : P, \mathcal{P} & \text{if } \mathsf{wait}(x) \notin \mathcal{P} \\
\text{[R-DONE]} & \Vdash t : \mathsf{done} & \longrightarrow_{\mathcal{D}} & \Vdash & \\
\text{[R-IF-1]} & \Vdash t : \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & \longrightarrow_{\mathcal{D}} & \Vdash t : P & \text{if } [\![e]\!] \neq 0 \\
\text{[R-IF-2]} & \Vdash t : \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & \longrightarrow_{\mathcal{D}} & \Vdash t : Q & \text{if } [\![e]\!] = 0 \\
\text{[R-FORK]} & \Vdash t : \mathsf{fork}\{Q\}P, \mathcal{P} & \longrightarrow_{\mathcal{D}} & \Vdash t : P, s : Q, \mathcal{P} & \\
\text{[R-NEW]} & \mathcal{H} \Vdash t : \mathsf{new}\ x\ \mathsf{in}\ P & \longrightarrow_{\mathcal{D}} & \mathcal{H}, x : \bullet, 0 \Vdash t : P & \\
\text{[R-CALL]} & \Vdash t : A(\overline{e}) & \longrightarrow_{\mathcal{D}} & \Vdash t : P\{[\![\overline{e}]\!]/\overline{x}\} & \text{if } A(\overline{x}) = P \in \mathcal{D}
\end{array}
$$

differ depending on whether or not there exists a process that is waiting for such notification. In [R-NFY-1], one waiting process is awoken and guarded by the appropriate number of acquisitions. In [R-NFY-2], the side condition $\mathsf{wait}(x) \notin \mathcal{P}$ means that $\mathcal{P}$ does not contain a process of the form $t : \mathsf{wait}(x, n).P$, implying that no process is currently suspended waiting for a notification on x. In this case, the notification is simply lost. Rules [R-DONE], [R-IF-*], [R-FORK], [R-NEW] and [R-CALL] model terminated processes, conditional processes, forks, object creation and procedure calls as expected. Notice that [R-FORK] has an implicit assumption stating that the name s of the process being created is fresh. This is because the composition $\mathcal{P}, \mathcal{P}'$ is well defined only provided that $dom(\mathcal{P}) \cap dom(\mathcal{P}') = \emptyset$. Similarly, rule [R-NEW] implicitly assumes that the object x being created is fresh.

We write $\Longrightarrow_{\mathcal{D}}$ for the reflexive, transitive closure of $\longrightarrow_{\mathcal{D}}$ and $\mathcal{H} \Vdash \mathcal{P} \not\longrightarrow_{\mathcal{D}}$ if there exist no $\mathcal{H}'$ and $\mathcal{P}'$ such that $\mathcal{H} \Vdash \mathcal{P} \longrightarrow_{\mathcal{D}} \mathcal{H}' \Vdash \mathcal{P}'$. With this notation in place, we formalize the property that we aim to ensure with the type system:

Definition 1 (deadlock-free program). *We say that* $(\mathcal{D}, P)$ *is* deadlock free *if* $\emptyset \Vdash main : P \Longrightarrow_{\mathcal{D}} \mathcal{H} \Vdash \mathcal{P} \not\longrightarrow_{\mathcal{D}}$ *implies* $\mathcal{H} = \{x_i : \bullet, 0\}_{i \in I}$ *and* $\mathcal{P} = \emptyset$.

In words, a program is deadlock free if every maximal, finite computation starting from the initial state – in which the heap is empty and there is only one running process $main : P$ – ends in a state in which all the objects in the heap are unlocked and all processes have terminated.

We conclude the section showing how to model the producer-consumer coordination program of Figure 1 in our calculus.

Example 1 (multiple-producers/single-consumer coordination). We model the program in Figure 1 by means of the following procedure definitions, which make use of two objects x and y to coordinate producer and consumer:

$$
\begin{aligned}
\mathtt{Main} &= \mathsf{new}\ x\ \mathsf{in}\ \mathsf{new}\ y\ \mathsf{in}\ \mathsf{acq}(x).\mathsf{acq}(y).\mathsf{fork}\{\mathtt{P}(x)\}\ \mathsf{fork}\{\mathtt{P}(y)\}\ \mathtt{C}(x, y) \\
\mathtt{P}(x) &= \mathsf{acq}(x).\mathsf{notify}(x).\mathsf{wait}(x).\mathsf{rel}(x).\mathtt{P}(x) \\
\mathtt{C}(x, y) &= \mathsf{wait}(x).\mathsf{notify}(x).\mathsf{wait}(y).\mathsf{notify}(y).\mathtt{C}(x, y)
\end{aligned}
$$

Table 3. Syntax of types and usages.

Type	$\mathtt{T} ::= \mathsf{int} \mid n \cdot U$
Usage	$U ::= \mathbf{0} \mid \kappa.U \mid U \mid V \mid U + V \mid \alpha \mid \mu\alpha.U$
Usage prefix	$\kappa ::= \mathit{acq} \mid \mathit{rel} \mid \mathit{wait} \mid \mathit{waiting} \mid \mathit{notify}$

The overall structure of these procedures matches quite closely that of the correspondings methods in Figure 1. Let us discuss the differences. First of all, in the calculus we focus on coordination primitives. All other operations performed on objects – notably, `Put` and `Get` in Figure 1 – that do not affect coordination are not modeled explicitly. Second, we model `while` loops using recursion. Third, we model `synchronized` blocks in Figure 1 as (matching) pairs of acquire-release operations on objects. Notice that the consumer never performs explicit releases, for x and y are always released by $\mathsf{wait}(x)$ and $\mathsf{wait}(y)$. This corresponds to the fact that, in Figure 1, the whole code of the consumer runs within a block that synchronizes on x and y.

It is worth noting that, even after all the background noise that is present in Figure 1 has been removed, understanding whether the program deadlocks is not trivial. As anticipated in the introduction, the most critical aspect is determining whether x and y are always notified at a time when there is a process waiting to be awoken. ■

3 Types and Usages

Our type system rules out deadlocks using two orthogonal mechanisms. The first one makes sure that well-typed programs do not contain cycles of parallel processes linked by shared objects. This mechanism suffices to avoid circular waits, but does not guarantee that each process suspended on a wait operation is awoken by a notification. To rule out these situations, we also associate each shared object with a simple protocol description, called *usage*, that specifies the operations performed by processes on it. Then, we make sure that usages are *reliable*, namely that each wait operation is matched by (at least) one notification. The rest of this section is devoted to the formal definition of types, usages and related notions, leading to the formalization of reliability. The description of the actual typing rules is deferred to Section 4.

The syntax of types and usages is shown in Table 3. A type is either int, denoting an integer value, or an object type $n \cdot U$ where n is a natural number called *counter* and U is a *usage*. The counter indicates the number of times the object has been acquired. The usage describes the combined operations performed by processes on the object. The usage $\mathbf{0}$ describes an object on which no operations are performed. The usage $\kappa.U$ describes an object that is used to perform the operation κ and then according to U. Usage prefixes *acq*, *rel*, *wait*, *waiting* and *notify* are in direct correspondence with the actions in Table 1. In particular, *waiting* is a "runtime version" of *wait* and describes an object on which the wait

operation has been performed and is waiting to be notified. Usages of user programs are not supposed to contain the *waiting* prefix. The usage $U \mid V$ describes an object that is used by concurrent processes according to U and V, whereas the usage $U + V$ describes an object that is used either according to U or to V. Terms of the form $\mu\alpha.U$ and α are used to describe recursive usages.

We adopt standard conventions concerning (recursive) usages: we assume contractiveness, namely that, in a usage $\mu\alpha.U$, the variable α occurring in U is always guarded by a usage prefix; we identify usages modulo (un)folding of recursions; we omit trailing $\mathbf{0}$'s.

To illustrate usages before describing the typing rules, consider the process

$$\mathsf{fork}\{\mathsf{acq}(x).\mathsf{rel}(x)\}\ \mathsf{acq}(y).\mathsf{acq}(x).\mathsf{rel}(x).\mathsf{rel}(y)$$

which uses two objects x and y. Notice that x is acquired by two concurrent subprocesses and that, in one case, this acquisition is nested within the acquisition and release of y. The operations performed on x are described by the usage $acq.rel \mid acq.rel$ whereas those performed on y are described by the usage $acq.rel$.

We now proceed to define a series of auxiliary notions related to types and usages and that play key roles in the type system and the proofs of its soundness. To begin with, we formalize a predicate on usages that allows us to identify those objects on which there is no process waiting for a notification:

Definition 2 (wait-free usage). *Let* $\mathsf{wf}(\cdot)$ *be the least predicate on usages defined by the axioms and rules below:*

$$\mathsf{wf}(\mathbf{0}) \qquad \mathsf{wf}(acq.U) \qquad \frac{\mathsf{wf}(U) \qquad \mathsf{wf}(V)}{\mathsf{wf}(U \mid V)} \qquad \mathsf{wf}(U + V)$$

Note that $\mathsf{wf}(U + V)$ holds regardless of U and V because the usage $U + V$ describes an object on which a process behaves according to either U or V, but the process has not committed to any such behavior yet. Hence, the process cannot be waiting for a notification on the object.

Next, we define a reduction relation $\rightsquigarrow$ describing the evolution of the type of an object as the object is used by processes. As we have anticipated earlier, the effect of operations depends on whether and how many times the object has been locked. For this reason, the reduction relation we are about to define concerns types and not just usages.

Definition 3 (type reduction). *Let* $\equiv$ *be the least congruence on usages containing commutativity and associativity of* $\mid$ *with identity* $\mathbf{0}$*, commutativity, associativity, and idempotency of* $+$*. The reduction relation* $\mathtt{T} \rightsquigarrow \mathtt{T}'$ *is the least relation defined by the axioms and rules in Table 4. As usual, we write* $\rightsquigarrow^*$ *for the reflexive and transitive closure of* $\rightsquigarrow$*.*

Rules [U-ACQ-1] and [U-REL-1] model the acquisition and the release of an object; in the first case, the object must be unlocked (the counter is 0) whereas, in the second case, the object must have been locked exactly once. Rules [U-ACQ-2] and

Table 4. Reduction of types.

$$[\text{U-ACQ-1}]\quad 0 \cdot acq.U \mid V \rightsquigarrow 1 \cdot U \mid V \qquad [\text{U-ACQ-2}]\quad n+1 \cdot U \rightsquigarrow n+2 \cdot U$$

$$[\text{U-REL-1}]\quad 1 \cdot rel.U \mid V \rightsquigarrow 0 \cdot U \mid V \qquad [\text{U-REL-2}]\quad n+2 \cdot U \rightsquigarrow n+1 \cdot U$$

$$[\text{U-WAIT}]\quad n+1 \cdot wait.U \mid V \rightsquigarrow 0 \cdot waiting.U \mid V \qquad [\text{U-CHOICE}]\quad n \cdot (U + U') \mid V \rightsquigarrow n \cdot U \mid V$$

$$[\text{U-NFY-1}]\quad n+1 \cdot waiting.U \mid notify.U' \mid V \rightsquigarrow n+1 \cdot acq.U \mid U' \mid V$$

$$[\text{U-NFY-2}]\quad \frac{\mathsf{wf}(V)}{n+1 \cdot notify.U \mid V \rightsquigarrow n+1 \cdot U \mid V} \qquad [\text{U-CONG}]\quad \frac{U \equiv U' \qquad n \cdot U' \rightsquigarrow n' \cdot V' \qquad V' \equiv V}{n \cdot U \rightsquigarrow n' \cdot V}$$

[U-REL-2] model nested acquisitions and releases on an object. These rules simply change the counter to reflect the actual number of (nested) acquisitions and do not correspond to an actual prefix in the usage. Rule [U-WAIT] models a wait operation performed on an object: the counter is set to 0 and the *wait* prefix is replaced by *waiting*, indicating that the object has been unlocked waiting for a notification. This makes it possible for another process to acquire the object and eventually notify the one who waits. Rules [U-NFY-1] and [U-NFY-2] model notifications. In [U-NFY-1], the notification occurs at a time when there is indeed another process that is waiting to be notified. In this case, the waiting process attempts to acquire the object. In [U-NFY-2], the notification occurs at a time when no other process is waiting to be notified. In this case, the notification has no effect whatsoever and is lost. Finally, rule [U-CHOICE] (in conjunction with commutativity of $+$) models the nondeterministic choice between two possible usages of an object and [U-CONG] closes reductions under usage congruence.

Not all types are acceptable for our type system. More specifically, there are two properties that we wish to be guaranteed:

- whenever there is a pending acquisition operation on a locked object, the object is eventually unlocked;
- whenever there is a pending wait operation on an object, the object is eventually notified.

A type that satisfies these two properties is said to be *reliable*:

Definition 4 (type reliability). *We say that* $\mathtt{T}$ *is* reliable*, written* $rel(\mathtt{T})$*, if the following conditions hold for all* n*,* U *and* V*:*

1. $\mathtt{T} \rightsquigarrow^* n \cdot acq.U \mid V$ *implies* $n \cdot V \rightsquigarrow^* 0 \cdot V'$ *for some* V'*, and*
2. $\mathtt{T} \rightsquigarrow^* 0 \cdot waiting.U \mid V$ *implies* $0 \cdot V \rightsquigarrow^* n \cdot notify.U' \mid V'$ *for some* U', V'*.*

For example, it is easy to verify that $0 \cdot acq.(wait.rel \mid acq.notify.rel)$ is reliable whereas $0 \cdot acq.wait.rel \mid acq.notify.rel$ is not. In the latter usage, the object may be acquired by a process that notifies the object at a time when there is no other process waiting to be notified. Eventually, the object is acquired again but the awaited notification is lost.

Example 2. Consider the type $0 \cdot acq.(U \mid V)$ where $U \stackrel{\text{def}}{=} \mu\alpha.acq.notify.wait.rel.\alpha$ and $V \stackrel{\text{def}}{=} \mu\alpha.wait.notify.\alpha$. To prove that $0 \cdot acq.(U \mid V)$ is reliable, we derive

$$\begin{array}{lll}
0 \cdot acq.(U \mid V) & \rightsquigarrow 1 \cdot U \mid V & \text{[U-ACQ-1]} \\
& \rightsquigarrow 0 \cdot U \mid waiting.notify.V & \text{[U-WAIT]}\ (\star) \\
& \rightsquigarrow 1 \cdot notify.wait.rel.U \mid waiting.notify.V & \text{[U-ACQ-1]} \\
& \rightsquigarrow 1 \cdot wait.rel.U \mid acq.notify.V & \text{[U-NFY-1]} \\
& \rightsquigarrow 0 \cdot waiting.rel.U \mid acq.notify.V & \text{[U-WAIT]} \\
& \rightsquigarrow 1 \cdot waiting.rel.U \mid notify.V & \text{[U-ACQ-1]} \\
& \rightsquigarrow 1 \cdot acq.rel.U \mid V & \text{[U-NFY-1]} \\
& \rightsquigarrow 0 \cdot acq.rel.U \mid waiting.notify.V & \text{[U-WAIT]} \\
& \rightsquigarrow 1 \cdot rel.U \mid waiting.notify.V & \text{[U-ACQ-1]} \\
& \rightsquigarrow 0 \cdot U \mid waiting.notify.V & \text{[U-REL-1]}\ (\star)
\end{array}$$

with the obvious applications of [U-CONG] not reported. Observe that no other reductions are possible apart from those shown above, that the two types labelled $(\star)$ are equal, and that both conditions of Definition 4 are satisfied for each reachable state. As we shall see in Example 3, the type $0 \cdot acq.(U \mid V)$ describes the behavior of the main thread of Example 1 with respect to each buffer. ■

4 Static Semantics

4.1 Type environments

The type system uses *type environments*, ranged over by Γ, which are finite sets of associations on variables and procedure names defined by the grammar below:

$$\textbf{Type environment} \quad \Gamma \quad ::= \quad \emptyset \mid x : \mathtt{T}, \Gamma \mid A : [\overline{\mathtt{T}}], \Gamma$$

An association $x : \mathtt{T}$ indicates that x has type $\mathtt{T}$, whereas an association $A : [\overline{\mathtt{T}}]$ indicates that A is a procedure accepting parameters of type $\overline{\mathtt{T}}$.

We write $dom(\Gamma)$ for the set of variable/procedure names for which there is an association in Γ and $\Gamma(x)$ for the type associated with $x \in dom(\Gamma)$. With an abuse of notation, we write Γ, Γ' for the union of Γ and Γ' when $dom(\Gamma) \cap dom(\Gamma') = \emptyset$. In addition: we write $live(\Gamma)$ for the subset of $dom(\Gamma)$ of *live object references* on which there are pending operations, that is $live(\Gamma) \stackrel{\text{def}}{=} \{x \in dom(\Gamma) \mid \Gamma(x) = n \cdot U \wedge (n > 0 \vee U \not\equiv \mathbf{0})\}$; we write $iszero(\Gamma)$ if $\Gamma(x) = 0 \cdot U$ for every $x \in dom(\Gamma)$; we write $noAct(\Gamma)$ if $live(\Gamma) = \emptyset$.

The same object may be used in different ways in different parts of a program. In order to track the combined usage of the object we inductively define two

operators $\mid$ and $+$ on type environments with the same domain. Intuitively, $(\Gamma \mid \Gamma')(x)$ is the type of an object that is used *both* as specified in Γ *and also* as specified in Γ' whereas $(\Gamma + \Gamma')(x)$ is the type of an object that is used *either* as specified in Γ *or* as specified in Γ'. The former case happens if x is shared by two concurrent processes respectively typed by Γ and Γ'. The latter case happens if x is used in different branches of a conditional process. Formally:

$$\begin{aligned}
\emptyset \mid \emptyset &= \emptyset \\
x : \mathsf{int}, \Gamma \mid x : \mathsf{int}, \Gamma' &= x : \mathsf{int}, (\Gamma \mid \Gamma') \\
A : [\overline{\mathtt{T}}], \Gamma \mid A : [\overline{\mathtt{T}}], \Gamma' &= A : [\overline{\mathtt{T}}], (\Gamma \mid \Gamma') \\
n \cdot U, \Gamma \mid m \cdot V, \Gamma' &= n + m \cdot U \mid V, (\Gamma \mid \Gamma') \quad n = 0 \vee m = 0
\end{aligned}$$

$$\begin{aligned}
\emptyset + \emptyset &= \emptyset \\
x : \mathsf{int}, \Gamma + x : \mathsf{int}, \Gamma' &= x : \mathsf{int}, (\Gamma + \Gamma') \\
A : [\overline{\mathtt{T}}], \Gamma + A : [\overline{\mathtt{T}}], \Gamma' &= A : [\overline{\mathtt{T}}], (\Gamma + \Gamma') \\
n \cdot U, \Gamma + n \cdot V, \Gamma' &= n \cdot U + V, (\Gamma + \Gamma')
\end{aligned}$$

Note that both $\mid$ and $+$ for environments are partial operators and that the former enforces the property that the same object cannot be owned by more than one process at any given time (at least one of the counters must be 0). It is easy to see that $\mid$ on environments is commutative and associative (modulo $\equiv$ on usages). In the following we write $\prod_{i=1..n} \Gamma_i$ in place of $\Gamma_1 \mid \cdots \mid \Gamma_n$.

As usual for behavioral type systems, the type environment used for typing a process is an abstraction of the behavior of the process projected on the objects it uses. In particular, a *live* object association $x : n \cdot U$ in the type environment of a process means that the process uses x as specified by U, whereas an association such as $x : 0 \cdot \mathbf{0}$ means that the process does not use x at all. To prevent circular waits between parallel processes, we forbid the existence of cycles in the corresponding type environments:

Definition 5 (acyclic type environments). *We say that a family $\{\overline{\Gamma}\}$ of type environments has a* cycle $x_1, \ldots, x_n$ *of $n \geq 2$ pairwise distinct names if there exist $\Gamma_1, \ldots, \Gamma_n \in \{\overline{\Gamma}\}$ such that $x_i \in \mathit{live}(\Gamma_i) \cap \mathit{live}(\Gamma_{(i \bmod n)+1})$ for all $1 \leq i \leq n$. We say that $\{\overline{\Gamma}\}$ is* acyclic *if it has no cycle.*

Observe that the acyclicity of a family of type environments can be established efficiently, for example by means of a standard union-find algorithm that stores in the same partition two environments if they share a live object.

4.2 Typing rules for user syntax

The typing rules for the language in Section 2 are defined in Table 5 and derive three kinds of judgments. A judgment $\Gamma \vdash e : \mathtt{T}$ means that the expression e is well typed in Γ and has type $\mathtt{T}$. A judgment $\Gamma \vdash P$ means that the process P is well typed in Γ. In particular, P uses each object $x \in \mathit{dom}(\Gamma)$ according to the usage in $\Gamma(x)$. Finally, a judgment $\Gamma \vdash (\mathcal{D}, P)$ means that the program $(\mathcal{D}, P)$ is well typed in Γ. We now describe the typing rules.

Table 5. Typing rules for user syntax.

Typing rules for (sequences of) expressions $\boxed{\Gamma \vdash \overline{e} : \overline{\mathtt{T}}}$

$$[\text{T-CONST}]\ \frac{noAct(\Gamma)}{\Gamma \vdash n : \mathsf{int}} \qquad [\text{T-VAR}]\ \frac{noAct(\Gamma)}{\Gamma, x : \mathtt{T} \vdash x : \mathtt{T}} \qquad [\text{T-OP}]\ \frac{\Gamma \vdash e : \mathsf{int} \qquad \Gamma \vdash e' : \mathsf{int}}{\Gamma \vdash e\ \mathsf{op}\ e' : \mathsf{int}}$$

$$[\text{T-SEQ}]\ \frac{\Gamma_i \vdash e_i : \mathtt{T}_i\ ^{(i=1..n)}}{\Gamma_1 \mid \cdots \mid \Gamma_n \vdash e_1, \ldots, e_n : \mathtt{T}_1, \ldots, \mathtt{T}_n}$$

Typing rules for processes $\boxed{\Gamma \vdash P}$

$$[\text{T-DONE}]\ \frac{noAct(\Gamma)}{\Gamma \vdash \mathsf{done}} \qquad [\text{T-ACQ-1}]\ \frac{\Gamma, x : 1 \cdot U \vdash P}{\Gamma, x : 0 \cdot acq.U \vdash \mathsf{acq}(x).P} \qquad [\text{T-ACQ-2}]\ \frac{\Gamma, x : n+2 \cdot U \vdash P}{\Gamma, x : n+1 \cdot U \vdash \mathsf{acq}(x).P}$$

$$[\text{T-REL-1}]\ \frac{\Gamma, x : 0 \cdot U \vdash P}{\Gamma, x : 1 \cdot rel.U \vdash \mathsf{rel}(x).P} \qquad [\text{T-REL-2}]\ \frac{\Gamma, x : n+1 \cdot U \vdash P}{\Gamma, x : n+2 \cdot U \vdash \mathsf{rel}(x).P}$$

$$[\text{T-WAIT}]\ \frac{\Gamma, x : n+1 \cdot U \vdash P}{\Gamma, x : n+1 \cdot wait.U \vdash \mathsf{wait}(x).P} \qquad [\text{T-NOTIFY}]\ \frac{\Gamma, x : n+1 \cdot U \vdash P}{\Gamma, x : n+1 \cdot notify.U \vdash \mathsf{notify}(x).P}$$

$$[\text{T-IF}]\ \frac{\Gamma \vdash e : \mathsf{int} \qquad \Gamma_i \vdash P_i\ ^{(i=1,2)}}{\Gamma \mid (\Gamma_1 + \Gamma_2) \vdash \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2} \qquad [\text{T-CALL}]\ \frac{\Gamma \vdash \overline{e} : \overline{\mathtt{T}}}{\Gamma, A : [\overline{\mathtt{T}}] \vdash A(\overline{e})}$$

$$[\text{T-FORK}]\ \frac{\Gamma_i \vdash P_i\ ^{(i=1,2)} \qquad iszero(\Gamma_1) \qquad \{\Gamma_1, \Gamma_2\}\ \text{acyclic}}{\Gamma_1 \mid \Gamma_2 \vdash \mathsf{fork}\{P_1\}P_2} \qquad [\text{T-NEW}]\ \frac{\Gamma, x : 0 \cdot U \vdash P \qquad rel(0, U)}{\Gamma \vdash \mathsf{new}\ x\ \mathsf{in}\ P}$$

Typing rule for programs $\boxed{\Gamma \vdash (\mathcal{D}, P)}$

$$[\text{T-PROGRAM}]\ \frac{\Gamma = A_i : [\overline{\mathtt{T}}_i]\ ^{(i=1..n)} \qquad \Gamma, \overline{x}_i : \overline{\mathtt{T}}_i \vdash P_i\ ^{(i=1..n)} \qquad \Gamma \vdash P}{\Gamma \vdash (\{A_i(\overline{x}_i) = P_i\}_{i=1..n}, P)}$$

The typing rules for (sequences of) expressions are unremarkable except for the fact that the unused part of the type environment cannot contain live associations, hence the premise $noAct(\Gamma)$ in [T-CONST] and [T-VAR].

Rule [T-DONE] states that the terminated process is well typed in an environment without live associations, because done does not perform any operation.

Rules [T-ACQ-1] and [T-ACQ-2] concern a process $\mathsf{acq}(x).P$ that acquires the lock on x and then continues as P. The difference between the two rules is that in [T-ACQ-1] the process is attempting to acquire the lock for the first time (the counter of the object is 0), whereas in [T-ACQ-2] the process is performing a reentrant acquisition, having already acquired the lock $n+1$ times. The continuation P is typed in an environment that reflects the (possibly reentrant) acquision of x. Note that the *acq* action occurs in the usage of x only in the case of [T-ACQ-1]. As we have anticipated in Section 3, in usages we only keep track of non-reentrant acquisitions and releases.

Rules [T-REL-1] and [T-REL-2] concern a process $\mathsf{rel}(x).P$. As in the case of [T-ACQ-*] rules, they differ depending on whether the lock is actually released ([T-REL-1]) or not ([T-REL-2]). Only in the first case the release action is noted in the usage of x. Besides that, the rules update the environment for typing the continuation P.

Rules [T-WAIT] and [T-NOTIFY] concern the coordination primitives. In both cases, the object must have been previously acquired (the counter is strictly positive) and the number of acquisitions does not change.

Rule [T-IF] is essentially standard. Since only one of the two continuations P_1 and P_2 executes, the respective type environments Γ_1 and Γ_2 are composed using the appropriate disjunctive operator.

Rule [T-FORK] types a parallel composition of two processes P_1 and P_2. The objects used by the parallel composition are used *both* by P_1 and also by P_2. For this reason, the respective type environments are combined using $|$. The $iszero(\Gamma_1)$ premise enforces the property that the process P_1 being forked off does not own any lock. The last premise requires that the family $\{\Gamma_1, \Gamma_2\}$ be acyclic, which is equivalent to checking that $live(\Gamma_1) \cap live(\Gamma_2)$ contains at most one element. This prevents circular waits between P_1 and P_2 as discussed earlier.

Rule [T-NEW] concerns the creation of a new object. The object is initially unlocked (its counter is 0) and its type must be reliable (Definition 4).

Rule [T-CALL] is unremarkable and types a process invocation. The only standard requirement is for the types of the arguments to match those expected in the corresponding process declaration.

The typing rule [T-PROGRAM] ensures that all process names have a corresponding definition and verifies that the main process is itself well typed.

Example 3. Let us show that the `Main` process in Example 1 is well typed. To do that, consider the type environment $\Gamma = \mathtt{P} : [0 \cdot U], \mathtt{C} : [1 \cdot V, 1 \cdot V]$ where U and V are the usages defined in Example 2 and observe that $noAct(\Gamma)$ holds.

For the two invocations $\mathtt{P}(x)$ and $\mathtt{P}(y)$ we easily derive

$$(1)\quad \frac{\overline{\Gamma, x : 0 \cdot U, y : 0 \cdot \mathbf{0} \vdash x : 0 \cdot U}}{\Gamma, x : 0 \cdot U, y : 0 \cdot \mathbf{0} \vdash \mathtt{P}(x)} \qquad (2)\quad \frac{\overline{\Gamma, x : 0 \cdot \mathbf{0}, y : 0 \cdot U \vdash y : 0 \cdot U}}{\Gamma, x : 0 \cdot \mathbf{0}, y : 0 \cdot U \vdash \mathtt{P}(y)}$$

using [T-VAR] and [T-CALL]. Then we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : 1 \cdot V, y : 1 \cdot V \vdash x, y : 1 \cdot V, 1 \cdot V}\,[\text{T-SEQ}]}{\Gamma, x : 1 \cdot V, y : 1 \cdot V \vdash \mathtt{C}(x, y)}\,[\text{T-CALL}] \quad (2)}{\Gamma, x : 1 \cdot V, y : 1 \cdot U \mid V \vdash \mathsf{fork}\{\mathtt{P}(y)\}\ \mathtt{C}(x, y)}\,[\text{T-FORK}] \quad (1)}{\Gamma, x : 1 \cdot U \mid V, y : 1 \cdot U \mid V \vdash \mathsf{fork}\{\mathtt{P}(x)\}\ \mathsf{fork}\{\mathtt{P}(y)\}\ \mathtt{C}(x, y)}\,[\text{T-FORK}]}{\Gamma, x : 1 \cdot U \mid V, y : 0 \cdot acq.(U \mid V) \vdash \mathsf{acq}(y).\mathsf{fork}\{\mathtt{P}(x)\}\ \mathsf{fork}\{\mathtt{P}(y)\}\ \mathtt{C}(x, y)}\,[\text{T-ACQ-1}]}{\Gamma, x : 0 \cdot acq.(U \mid V), y : 0 \cdot acq.(U \mid V) \vdash \mathsf{acq}(x) \cdots}\,[\text{T-ACQ-1}]}{\Gamma, x : 0 \cdot acq.(U \mid V) \vdash \mathsf{new}\ y\ \mathsf{in}\ \cdots}\,[\text{T-NEW}]}{\Gamma \vdash \mathsf{new}\ x\ \mathsf{in}\ \mathsf{new}\ y\ \mathsf{in}\ \mathsf{acq}(x).\mathsf{acq}(y).\mathsf{fork}\{\mathtt{P}(x)\}\ \mathsf{fork}\{\mathtt{P}(y)\}\ \mathtt{C}(x, y)}\,[\text{T-NEW}]$$

where the reliability of $0 \cdot acq.(U \mid V)$ needed in the applications of [T-NEW] has already been proved in Example 2. In the applications of [T-FORK], the acyclicity of the involved environments is easily established since in each conclusion of (1) and (2) there is only one live association, for x and y respectively. ■

4.3 Typing rules for runtime syntax and states

The soundness proof of our type system follows a standard structure and includes a subject reduction result stating that typing (but not necessarily types) are preserved by reductions. Since the operational semantics of a program makes use of constructs that occur at runtime only (notably, waiting processes and states) we must extend the typing rules to these constructs before we can formulate the properties of the type system. The additional typing rules are given in Table 6.

Rule [T-WAITING] accounts for a process waiting for a notification, after which it will attempt to acquire the lock on x. Once x is notified and the process awakened, the process will acquire the lock n times, reflecting the state of acquisitions at the time the process performed the $\mathsf{wait}(x)$ operation (see [R-WAIT]).

The rule [T-STATE] for states looks more complex than it actually is. For the most part, this rule is a generalization of [T-FORK] and [T-NEW] to an arbitrary

Table 6. Typing rules for runtime syntax.

$$[\text{T-WAITING}]\quad \dfrac{\Gamma, x : n \cdot U \vdash P}{\Gamma, x : 0 \cdot waiting.U \vdash \mathsf{wait}(x, n).P}$$

$$[\text{T-STATE}]\quad \dfrac{\Gamma = \prod_{i \in I} \Gamma_i \qquad \Gamma_i \vdash P_i \ ^{(i \in I)} \qquad rel(\Gamma(x_j))\ ^{(j \in J)} \qquad \{\Gamma_i\}_{i \in I}\ \text{acyclic} \qquad \Gamma_i(x_j) = n + 1, U \iff t_i = s_j\ ^{(i \in I, j \in J)} \qquad \Gamma(x_j) = 0 \cdot U \iff s_j = \bullet\ ^{(j \in J)}}{\Gamma \vdash \{x_j : s_j, n_j\}_{j \in J} \Vdash \prod_{i \in I} t_i : P_i}$$

number of concurrent processes (indexed by $i \in I$) and of objects (indexed by $j \in J$). From left to right, the premises of the rule ensure that:

- each process P_i is well typed in its corresponding environment Γ_i;
- the type of each object x_j, which describes the overall usage of x_j by all the processes, is reliable;
- the family $\{\Gamma_i\}_{i \in I}$ of type environments is acyclic;
- the process t_i owns the object x_j if and only if the counter for x_j in the type environment Γ_i of P_i is strictly positive. Because of the definition of $|$ for type environments, this implies that no other process owns x_j;
- no process owns the object x_j if and only if the counter for x_j in the overall environment is zero.

4.4 Properties of well-typed programs

As usual, the key lemma for proving soundness of the type system is subject reduction, stating that a well-typed state reduces to well-typed state. In our case, this result guarantees the preservation of typing, but not necessarily the preservation of types. Indeed, as a program reduces and operations are performed on objects, the type of such objects changes consequently. To account for these changes, we lift reduction of types to type environments, thus:

Definition 6 (environment reduction). *The reduction relation for environments, noted $\rightsquigarrow$, is the least relation such that:*

$$\Gamma \rightsquigarrow \Gamma, x : \mathtt{T} \qquad \frac{\mathtt{T} \rightsquigarrow \mathtt{T}'}{\Gamma, x : \mathtt{T} \rightsquigarrow \Gamma, x : \mathtt{T}'}$$

As usual, $\rightsquigarrow^*$ denotes the reflexive, transitive closure of $\rightsquigarrow$.

The first rule accounts for the possibility that a new object x is created. We can now formally state subject reduction, which shows that typing is preserved for any reduction of a well-typed program:

Lemma 1 (subject reduction). *Let $\Gamma \vdash (\mathcal{D}, P)$ and $\emptyset \Vdash main : P \longrightarrow_{\mathcal{D}}^* \mathcal{H}' \Vdash \mathcal{P}'$. Then $\Gamma' \vdash \mathcal{H}' \Vdash \mathcal{P}'$ for some Γ' such that $\Gamma \rightsquigarrow^* \Gamma'$.*

The soundness theorem states that well-typed programs are deadlock free:

Theorem 1 (soundness). *If $\Gamma \vdash (\mathcal{D}, P)$, then $(\mathcal{D}, P)$ is deadlock free.*

5 Related work

Despite the number of works on deadlock analysis of concurrent programs, only a few address coordination primitives. Below we discuss the most relevant ones.

Static techniques typically employ control-flow analysis to build a dependency graph between objects and enforce its acyclicity. These techniques may

adopt some heuristics to remove likely false positives, but they are necessarily conservatives. For instance, Deshmukh *et al.* [5] analyze libraries of concurrent objects looking for deadlocks that may manifest for *some* clients of such objects, by considering all possible aliasing between the locks involved in the objects. The technique of von Praun [17] is based on the detection of particular patterns in the code, such as two threads that perform $\mathsf{wait}(x)$ and $\mathsf{wait}(y)$ in different orders, which does not necessarily lead to a deadlock. Naik *et al.* [16] combine different static analyses that correspond to different conditions that are necessary to yield a deadlock. Their technique concerns lock acquisition and release, but not coordination primitives. Williams *et al.* [19] build a *lock-order graph* that describes the sequences of lock acquisitions in a library of Java classes. In particular, they consider implicit acquisitions due to wait operations, but not deadlocks caused by missed notifications. Hamin and Jacobs [8] present a refinement of separation logic to reason on locks and wait/notify coordination primitives. Their technique ensures deadlock freedom by imposing an ordering on the use of locks and by checking that each wait is matched by at least one notification. The logic allows them to address single wait operations within loops, which is something our type system is unable to handle. On the other hand, the use of a lock ordering limits the technique in presence of loops and recursion whereby blocking operations on several locks are interleaved, as in our running example (Figure 1).

Dynamic techniques perform deadlock detection by analyzing the log or scheduling of a program execution [1, 11, 6]. By considering actual program runs, these techniques potentially offer better precision, at the cost of delayed deadlock detection. Agarwal and Stoller [1] define feasible sequences, called *traces*, that are consistent with the original order of events from each thread and with constraints imposed by synchronization events. By analyzing all the possible traces, they verify that a wait operation always *happens before* a notify operation. Joshi *et al.* [11] extract a simple multi-threaded program from the source code that records relevant operations for finding deadlocks. Then, they consider any possible interleavings of the simple program by means of a model checker in search of deadlocks. The technique returns both false positives (the simple program manifests a deadlock that never occurs in the source program) and false negatives (the simple program is defined by observing a single execution and the deadlock may occur in another execution). Deadlocks due to coordination primitives are not covered by the technique of Eslamimehr and Palsberg [6].

Demartini *et al.* [4] translate `Java` into the Promela language, for which the SPIN model checker verifies deadlock freedom. Their analysis reports all deadlock possibilities so long as the program does not exceed the maximum number of modeled objects or threads. `Java` Pathfinder, a well-known tool that is used to analyze execution traces, also performs model checking by translating `Java` to Promela [9]. When checking matches between wait and notify operations, this technique may require the analysis of a large number of traces.

Kobayashi and Laneve [13] present a deadlock analysis for the pi-calculus which provided the initial inspiration for this work. In fact, attempts were made to encode the language of Section 2 in the pi-calculus so as to exploit the tech-

nique of Kobayashi and Laneve. The encoding approach proved to be unsatisfactory because of the many false positives it triggered. Notably, Kobayashi and Laneve [13], following [12], define a notion of usage reliability which can be reduced to a reachability problem in Petri nets. However, the encoding of usages with `wait-notify` primitives requires the use of Petri nets with *inhibitor arcs* [2], which are more expressive than standard Petri nets.

Our type system does not impose an order on the usage of locks. Rather, it adopts a typing rule for parallel threads ([T-FORK]) inspired by session type systems based on linear logic [18, 3]. The key idea is to require that, whenever two threads are combined together in a parallel composition, they can only interact through at most one object, as suggested by the structure of the cut and tensor rules in (classical) linear logic [7]. This approach results in a simple and expressive type system which can deal with recursive processes interleaving blocking actions on different objects (Figure 1). The downside is that [T-FORK] imposes well-typed programs to exhibit a forest-like topology, ruling out some interesting programs which are in the scope of other techniques, such as those based on lams [13].

6 Concluding Remarks

We have described a deadlock analysis technique for concurrent programs using `Java`-like coordination primitives `wait` and `notify`. Our technique is based on behavioral types, called usages, that may be encoded as Petri nets with inhibitor arcs [2]. Thereby, we reduce deadlock freedom to the reachability problem in this class of Petri nets that is *partially decidable*. The details, as well as the extension of our technique to the `notifyAll` primitive can be found in the full paper [15].

Our technique is unable to address programs where two threads share more than one object because of the acyclicity constraint in rule [T-FORK]. A more fine-grained approach for tracking object dependencies has been developed by Laneve [14] and is based on lams [13]. However, this approach does not consider coordination primitives. We initially tried to combine lams and usages with coordination primitives, but the resulting type system proved to be overly restrictive with respect to recursive processes: the amount of dependencies prevented the typing of any recursive process interleaving blocking operations on two or more objects (such as the consumer in Figure 1). Whether lams and usages with coordination primitives can be reconciled is still to be determined.

Another limitation of the type system is that it assumes precise knowledge of the number of acquisitions for each shared object, to the point that types contain a *counter* for this purpose. However, this information is not always statically available. It may be interesting to investigate whether this limitation can be lifted by allowing a form of *counter polymorphism*.

Finally, our model ignores object fields and methods for the sake of simplicity. Coping with these features might require some non-trivial extensions of the type system, such as effect annotations on method types, in order to keep track of the type of fields as they are updated.

References

1. Agarwal, R., Stoller, S.D.: Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In: Proceedings of PADTAD'06. pp. 51–60. ACM (2006). https://doi.org/10.1145/1147403.1147413
2. Busi, N.: Analysis issues in petri nets with inhibitor arcs. Theor. Comput. Sci. **275**(1-2), 127–177 (2002). https://doi.org/10.1016/S0304-3975(01)00127-X
3. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016). https://doi.org/10.1017/S0960129514000218
4. Demartini, C., Iosif, R., Sisto, R.: A deadlock detection tool for concurrent java programs. Softw., Pract. Exper. **29**(7), 577–603 (1999)
5. Deshmukh, J.V., Emerson, E.A., Sankaranarayanan, S.: Symbolic modular deadlock analysis. Autom. Softw. Eng. **18**(3-4), 325–362 (2011)
6. Eslamimehr, M., Palsberg, J.: Sherlock: scalable deadlock detection for concurrent programs. In: Proceedings of FSE'14. pp. 353–365. ACM (2014)
7. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
8. Hamin, J., Jacobs, B.: Deadlock-free monitors. In: Ahmed, A. (ed.) Proceedings of ESOP'18. pp. 415–441. LNCS 10801, Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_15
9. Havelund, K.: Using runtime analysis to guide model checking of java programs. In: Proceedings of SPIN'00. pp. 245–264. LNCS 1885, Springer (2000). https://doi.org/10.1007/10722468_15
10. Hoare, C.A.R.: Monitors: An operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974). https://doi.org/10.1145/355620.361161
11. Joshi, P., Naik, M., Sen, K., Gay, D.: An effective dynamic analysis for detecting generalized deadlocks. In: Proceedings of FSE'10. pp. 327–336. ACM (2010). https://doi.org/10.1145/1882291.1882339
12. Kobayashi, N.: Type-based information flow analysis for the pi-calculus. Acta Informatica **42**(4-5), 291–347 (2005). https://doi.org/10.1007/s00236-005-0179-x
13. Kobayashi, N., Laneve, C.: Deadlock analysis of unbounded process networks. Inf. Comput. **252**, 48–70 (2017). https://doi.org/10.1016/j.ic.2016.03.004
14. Laneve, C.: A lightweight deadlock analysis for programs with threads and reentrant locks. In: Science of Computer Programming (to appear). A preliminary version appears in Proc. of FM'18. pp. 608–624. LNCS 10951, Springer (2018)
15. Laneve, C., Padovani, L.: Deadlock Analysis of Wait-Notify Coordination. Tech. rep., Computer Science Department, University of Bologna (2019), https://hal.inria.fr/hal-02166082
16. Naik, M., Park, C., Sen, K., Gay, D.: Effective static deadlock detection. In: Proceedings of ICSE'09. pp. 386–396. IEEE (2009). https://doi.org/10.1109/ICSE.2009.5070538
17. von Praun, C.: Detecting synchronization defects in multi-threaded object-oriented programs. Ph.D. thesis, Swiss Federal Institute of Technology, Zurich (2004)
18. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2-3), 384–418 (2014). https://doi.org/10.1017/S095679681400001X
19. Williams, A.L., Thies, W., Ernst, M.D.: Static deadlock detection for Java libraries. In: Proceedings of ECOOP'05. pp. 602–629. LNCS 3586, Springer (2005). https://doi.org/10.1007/11531142_26