An automatic technique for static deadlock prevention

Bertrand Meyer

ETH Zurich, Innopolis University & Eiffel Software

se.ethz.ch, university.innopolis.ru/en/research/selab/, eiffel.com

Abstract. Deadlocks remain one of the biggest threats to concurrent programming. Usually, the best programmers can expect is dynamic deadlock detection, which is only a palliative. Object-oriented programs, with their rich reference structure and the resulting presence of aliasing, raise additional problems. The technique developed in this paper relies on the "alias calculus" to offer a completely static and completely automatic analysis of concurrent object-oriented programs. The discussion illustrates the technique by applying it to two versions of the "dining philosophers" program, of which it proves that the first is deadlock-free and the second deadlock-prone.

1 Overview

Deadlock is, along with data races, one of the two curses of concurrent programming. Although other dangers — priority inversion, starvation, livelock — await concurrent programmers, these two are the most formidable. The goal of the technique presented here is to avoid deadlock entirely through static analysis, so that any program that could cause deadlock will be rejected at compile time. No implementation is available and the description is preliminary, leaving several problems open.

The general approach is applicable to any concurrency model, but its detailed application relies on SCOOP (Simple Concurrent Object-Oriented Programming) [3] [4] [7] [8] [9], a minimal concurrent extension to an OO language, retaining the usual modes of reasoning about sequential programs. One of the distinctive properties of SCOOP is that the model removes the risk of data races, but deadlocks are still possible. The goal of the approach described here is to remove deadlocks too, statically.

In today's practice, the best concurrent programmers may usually hope for is *dynamic* deadlock detection: if at run time the system runs into a deadlock, a watchdog will discover the situation and trigger an exception. Such a technique is preferable to letting the execution get stuck forever, but it is still unsatisfactory: the time of program execution is too late for detection. We should aim for *static* prevention through a technique that will analyze the program text and identify possible run-time deadlocks.

Like many interesting problems in programming, static deadlock detection is undecidable, so the best we can expect is an over-approximation: a technique that will flag all programs that might deadlock — meaning it is *sound* — but might occasionally flag one that won't. The technique should be as *precise* as possible, meaning that the number of such false alarms is minimal. In fact an unsound technique may be of interest too, if it detects many — but not all — deadlock risks. The technique described here is intended to be sound, but no proof of soundness is available.

The approach relies on two key ideas. The first idea is that deadlock prevention means finding out if there is any set of processors whose *hold sets* and *wait sets* are mutually non-disjoint. The second idea is that in an object-oriented context, with references and hence possible aliasing, we can compute these sets by applying *alias analysis* to get a model of the processors associated with concurrent objects, and hence of their hold and wait sets. The key supporting tool in this step is the **alias calculus**, a technique developed by the author and colleagues for fully automatic alias analysis. Section 7 is a hands-on application of the resulting technique to two programs implementing solutions to the well-known "dining philosophers" problem; the analysis proves that the first version — the standard SCOOP solution of this problem — is deadlock-free, and that the second version, specifically contrived to cause potential deadlocks, can indeed result in a deadlocked execution.

The discussion begins with a general formalization of the *deadlock condition*, applicable to any concurrency framework (section 2). Based on this model, a general strategy is possible for detecting deadlock statically; the principle is that deadlock may arise if the "hold sets" and "wait sets" of a set of processors are not pair-wise disjoint. This strategy is the topic of section 3. After a short reminder on SCOOP in section 4, section 5 shows how to produce a deadlock in SCOOP; the design of the model makes such a situation rare, but not impossible. Section 6 refines the general deadlock detection technique of section 3 to the specific case of SCOOP, showing the crucial role of alias analysis, as permitted by the alias calculus. Section 7 shows the application to an important and representative example problem: dining philosophers, in the case of two components. Section 8 lists the limitations of the present state of the work and the goals of its future development.

Although the literature on deadlock prevention and detection is huge, there is (to my knowledge) no precedent for an approach that, as presented here, permits static deadlock analysis for concurrent object-oriented programs by relying on alias analysis. This is the reason for the restricted nature of the bibliography.

2 General deadlock condition

Deadlock is, as mentioned above, only one of two major risks in traditional concurrent programming. It is closely connected to the other one, data races. A data race arises when two concurrent program elements access and modify data in an order that violates their individual assumptions; for example, if each tries to book a flight by first finding a seat then booking it, some interleavings of these operations will cause both to believe they have obtained a given seat. The remedy is to obtain exclusive access to a resource for as long as needed; but then, if concurrent elements share more than one resource and they obtain exclusive access through locking, the possibility of *deadlock* looms: the execution might run into a standstill because every element is trying to obtain a resource that one of the others has locked. In the flight example, one client might try to lock the seat list then the reservation list, and the other might try to lock them in the reverse order, bringing them to an endless "deadly embrace", as deadlocks are also called. This analysis indicates the close connection between the two plagues: to avoid data races, programmers lock resources; but the ability of multiple clients to lock multiple resources may lead to deadlock.

The two problems are, however, of a different nature. One may blame data races on the low level of abstraction of the usual concurrent programming techniques (such as threading libraries with synchronizaation through semaphores); the SCOOP model removes the risk of data races by requiring program elements to obtain exclusive access before using any shared resource. The key mechanism (section 4) is the SCOOP idiom for reserving *several* resources at once, moving the task of data race avoidance from the programmer to the SCOOP implementation. As a consequence, many deadlock cases disappear naturally. But, as we will see in detail, deadlock does remain possible, and is a harder problem to eliminate statically.

Ignoring SCOOP for the moment, we will now study under what general conditions deadlock can arise. The term "processor" will denote (as in SCOOP but without loss of generality) a mechanism able to execute *sequential* computations. Concurrency arises when more than one processor is at work. Processors can be of different kinds, hardware or software; a typical example is a *thread* as provided by modern operating systems.

The deadlock scheme considered here is the "Coffman deadlock", which assumes that two or more processors need exclusive access to two or more shared resources, and all seek to obtain it through locking. Deadlock arises if at some time during execution these processors become involved in a cycle, such that every one of them is seeking to lock a resource that is held by the next processor in the cycle. This is the usual informal definition, which we may formalize (without making the cycle explicit) as follows. There is a set P of processors and a set R of resources, both finite. For each processor p, at any execution time t, there are two disjoint sets of resources:

- $H(p)$, the hold (or "has") set, containing resources that p has locked and not yet unlocked, and to which, as a result, it has exclusive access.
- $W(p)$, the wait (or "want") set, containing resources that p is trying, unsuccessfully so far, to lock.

(To avoid ambiguity, we may make the time explicit, writing $W_t(p)$ and $W_t(p)$.)

Deadlock arises between the processors in P when every one of them wants something that another has:

$$\forall\, p: P \mid \exists\, p': P \mid W(p) \cap H(p') \neq \varnothing \qquad \text{-- Deadlock condition}$$

To get the usual cycle based presentation it suffices to start from an arbitrary processor p and follow the successive p' of the condition. Since P is finite the sequence is cyclic.

It is also useful to state the reverse condition, deadlock-freedom:

$$\exists\, p: P \mid \forall\, p': P \mid W(p) \cap H(p') = \varnothing \qquad \text{-- Progress condition}$$

This condition holds in particular when $W(p)$ is empty, that is to say, p is progressing normally and not waiting for any resource. It also holds if every processor only ever needs one resource at a time, formally expressed as $\forall\, p: P \mid H(p) \neq \varnothing \Rightarrow W(p) = \varnothing$: in that case, if the deadlock condition held, the cyclic sequence obtained by the above construction would consist of processors for which neither H nor W is empty, which contradicts this assumption.

3 Deadlock prevention strategy

The preceding analysis leads to a general strategy for statically detecting possible deadlocks. The strategy as presented here applies to any concurrency model; the next section will describe its application to the specific case of SCOOP.

Two observations are necessary to evaluate programs for their susceptibility to the deadlock condition. First, the condition refers to processors; but processors may only be known at run time. We need to transpose the reasoning to what we can analyze statically: positions in the program text. It suffices to extend the notations $H(p)$ and $W(p)$ to p denoting a program position; they denote the run-time hold and wait sets of any processor whose execution reaches position p.

The second observation is that it is only necessary to evaluate the condition at "locking positions": program points that contain an instruction that tries to lock a resource. Locking positions mark where deadlock can occur.

The general strategy, then, is to develop techniques for:

1 • Estimating the H and W sets of any locking position. (The technique may be more general, and yield these sets for any program position.)

2 • Estimating, for every locking position $lp1$, the set of its "*simultaneous*" positions: all locking positions $lp2$ such that during execution a processor may reach $lp1$ and another $lp2$ at the same time. Note that $lp1$ and $lp2$ may be the same position if several processors execute the same code.

3 • Computing $W(lp1) \cap H(lp2)$ for every simultaneous pair $[lp1, lp2]$.

The progress condition holds if this intersection is empty for at least one pair.

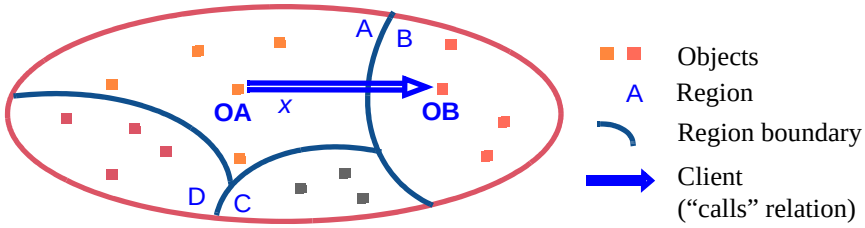
For the first two steps, the strategy "estimates" the result since it may not be possible to determine them exactly. As noted in section 1, an estimation should normally be an over-approximation, as accurate as possible.

The implementation of these two steps, and the precision of the estimation, depend on the concurrency model. We now come to the application to a specific model, SCOOP.

4 SCOOP basics

The SCOOP model simplifies the framework developed above. "Processor" is a central notion in SCOOP, and is general enough to subsume the concept of resource.

SCOOP closely connects the concurrency architecture with the object-oriented architecture of a system, through the rule that every object has an associated processor, its *handler*, which is responsible for all operations on the object (all calls of the form $x.r(\ldots)$ where r is attached to the object). The result is a partitioning of the object space into a number of regions, each associated with a handling processor:



In the figure, the object OA has a field *x*, attached (at a particular time during execution) to an object OB in another region. A call *x*.*r* (…) issued by the processor of region A (handler of OA) will be executed by the processor of region B (handler of OB).

Such a call applied to a separate object is (in the case of a procedure, rather than a function) **asynchronous**: it does not block the *A* processor. To reflect this property, a variable such as *x* that may represent objects in another region must be declared **separate**. But *x*.*r* (…) is only permitted if the processor executing this operation has obtained exclusive access to the object. The basic way to achieve exclusive access is through a procedure call with *x* as argument. More generally, a routine with header

```
r (x: separate T; y: separate U; …)
```

will guarantee exclusive access to the separate arguments for the duration of the execution of *r*'s body. A call *r* (*a*, *b*, …) may as a consequence have to wait until it has obtained exclusive access to the objects attached to *a*, *b*, …. This is how SCOOP gets rid of data races: if you need a resource, or any number of resources, you must first obtain exclusive access to them; so a program element cannot invalidate another's assumption by messing up with shared resources.

If at the time of such a call one of the actual arguments, for example *a*, is already accessible under exclusive access, perhaps because *a* is a formal argument of the routine in which the call appears, this exclusive access is transferred to *r*, under "lock passing", for the duration of its execution.

As an example of simultaneous reservation of multiple resources, the following code implements the "dining philosophers" scheme. Two classes are involved, *FORK* and *PHILOSOPHER*. All we need from a fork is the ability to help us eat:

```
class FORK feature
        use do …Use the fork to eat …end
end
```

The creation procedure (constructor) *make* of class *PHILOSOPHER* gives a philosopher a left and a right forks, represented as separate objects. The procedure *live* capture a philosopher's behavior:

```
class PHILOSOPHER create make feature
        left, right: separate FORK
```

```
        live
                        -- Perform philosopher cycle with left and right forks.
            do
                from until False loop
                    think
                    eat (left, right)          -- Simultaneous point (see section 7
                end
            end
        eat (l, r: separate FORK)
                        -- Pick both l and r.
            do
                l.use ; r.use
            end
        think do … Not specified … end

    feature -- Initialization
        make (l, r: separate FORK)
                        -- Initialize with l as left and r as right forks.
            do left := l ; right := r end
    end
```

The key element is the highlighted call *eat* (*left*, *right*) which, by virtue of the basic SCOOP processor reservation mechanism, obtains exclusive access to the two forks. No other synchronization operation is necessary. The classical problems of securing multiple resources without risking deadlock or starvation are no longer the application programmer's business; the SCOOP implementation handles them.

To set everything going we may use — in the illustrative case of two philosophers and two forks — the following "root" class. (The class text is supporting code, with no particularly deep concepts, but needed for the example analysis of the next sections):

```
class MEAL create make feature
    fork1, fork2: separate FORK
    phil1, phil2: separate PHILOSOPHER

    make
                    -- Set up two philosophers sharing forks, and get them started.
        do
                    -- Create the philosophers and forks:
            create fork1 ; create fork2
            create phil1.make (fork1, fork2)
            create phil2.make (fork2, fork1)
                    -- Start the philosophers:
            execute (phil1, phil2)
        end
```

```
feature {NONE}
        execute (p1, p2: separate PHILOSOPHER)
                        -- Perform both philosophers' lives.
                do
                        p1.live
                        p2.live
                end

end
```

The creation instructions for the philosophers reverse the "left" and "right" role for the two forks. It would not be possible to merge the creation procedure *make* with *execute* since it needs to perform calls such as *p1.live* on attached targets, requiring exclusive access to the corresponding objects; so we first need the creation instructions in *make* to create these objects, then *execute* to work on them.

In the version above, *make* does call *execute*, so that creating a meal object (**create** *m.make*) is enough to trigger a full system execution. As a result, *execute* is secret (private), as specified by **feature** {*NONE*}. It would also be possible to separate the two parts, removing the call to *execute* in *make* and declaring *execute* public.

The illustrated constructs are the main components of the SCOOP concurrency model. They suffice for the rest of the presentation, but for completeness it is useful to list the other properties which together with them make up all of SCOOP:

- A separate call on a *query* (function or attribute, returning a result), as in the assignment *y* := *x.f* (…) is, unlike a call to a *procedure*, synchronous: since the execution needs the value, it will wait until *f* has terminated. Waiting on the result of a query is the SCOOP mechanism for re-synchronizing after an asynchronous call.
- A routine precondition of the form **require** *x.some_condition*, where *x* is a separate formal argument of the routine, will cause the execution of the routine's body to wait until the condition is satisfied. This is the SCOOP mechanism for condition synchronization.
- To work on separate targets, a program element must have exclusive access to them. The usual way to obtain it is to pass the targets as arguments to a routine, as in *execute* (*phil1*, *phil2*). If *execute* is not available separately from *make*, it is possible to avoid writing *execute* altogether and replace the call *execute* (*phil1*, *phil2*) by the "inline separate" construct

  ```
  separate phil1 as p1, phil2 as p2 do
      p1.live ; p2.live
  end
  ```

  with the same effect of guaranteeing exclusive access to *phil1* and *phil2* under the local names *p1* and *p2*. Inline separate avoids the writing of wrapper routines for the sole purpose of performing simple operations on separate targets. The rest of the discussion will use explicit wrapping, with no loss of generality since object reservation and access has the same semantics with inline separate as with wrapping.

5 Deadlock in SCOOP

To discuss deadlock in the SCOOP context, we do not need to distinguish between processors and resources as in the general model introduced at the beginning of this discussion. As illustrated by philosophers and forks, the notion of processor covers resources as well. A resource is an object; exclusive access to it means exclusive access to its handling processor. This unification of concepts significantly simplifies the problem.

In the practice of SCOOP programming, many deadlock risks disappear thanks to SCOOP's signature mechanism of reserving several separate targets at once by passing them as arguments to a routine (or, equivalently, using an inline separate). The most common case of deadlock other approaches arises when *p* and *q* both want to reserve *r* and *s*, but attempt to lock them in reverse order, ending up with *p* having *r* and wanting *s* while *q* has *s* and wants *r*. In SCOOP they will both execute *some_routine* (*r*, *s*); each will proceed when it gets both. No deadlock is possible. (In addition the SCOOP implementation guarantees fairness.)

Unfortunately, even though deadlock will not normally arise in proper use of SCOOP, it remains a distinct theoretical possibility. It is in fact easy to construct examples of programs that may deadlock, such as the following variant of the dining philosophers solution. The correct eating procedure in *PHILOSOPHER*, repeated here for convenience, is

```
eat (l, r: separate FORK)
            -- Pick both l and r.
        do
            l.use r.use
        end
```

Now consider the following:

```
                                -- Features to be added to class PHILOSOPHER
eat_bad (f: separate FORK)
            -- Pick f, then the other fork.
        do
            pick_one (f)
            pick_second (f, opposite (f))
        end
pick_one (f: separate FORK)
            -- Pick f.
        do
            f.use
        end
```

```
pick_second (f1, f2 : separate FORK)
            -- Already holding f1, pick f2.
        do
            f2.use
        end
opposite (f : separate FORK)
            -- The fork other than f.
            -- (With more than two forks, would be replaced by a "next" function.)
        do
            Result := if f = left then right else left end
        end
```

and replace the key call *eat* (*left*, *right*) in procedure *live* by *eat_bad* (*left*) (or *eat_bad* (*right*)).

The need for procedure *pick_second* comes from the assumptions of the dining philosophers problem: if we replaced the second instruction of *eat_bad* with just *pick_one* (*opposite* (*f*)), no deadlock would arise, but we would violate the basic condition that a philosopher requires access to both forks at once. The first argument of *pick_second* serves to maintain hold on the first fork.

If this scheme seems convoluted it is precisely because deadlock does not naturally arise in ordinary SCOOP style. With this version, however, classic dining-philosophers deadlock is possible, with a run-time scenario such as this: *eat_bad* for *phil*1 executes *pick_one* (*fork*1); *eat_bad* for *phil*2 executes *pick_one* (*fork*2); then the first *eat_bad* tries to execute *pick_second* (*fork*1, *fork*2) and waits because *phil*2 holds *fork*2; but the second processor is also stuck, trying to execute *pick_second* (*fork*2, *fork*1) while *phil*1 holds *fork*1.

At this stage the deadlock condition of section 2 holds. Identifying each processor by the program name of an object it handles:

- H (*phil*1) = {*fork*1)
- W (*phil*1) = {*fork*2}
- H (*phil*2) = {*fork*2}
- W (*phil*2) = {*fork*1}
- Hence both H (*phil*1) $\cap$ W (*phil*2) $\neq \varnothing$ and H (*phil*2) $\cap$ W (*phil*1) $\neq \varnothing$.

6 The SCOOP deadlock detection rule

To apply the strategy of section 3, we must:

- Find which pairs of locking positions are "simultaneous" (that is to say, might be executed concurrently).
- For every such pair [*lp*, *lp'*] compute the hold and wait sets of *lp* and *lp'*.
- Find out if all the intersections H (*lp*) $\cap$ W (*lp'*) are non-empty.

In the SCOOP context, a locking position in SCOOP is a call *r* (*a*, *b*, …) to a routine with separate arguments. (As noted, we ignore inline separate instructions, which can be handled in the same way.)

We can define instruction simultaneity thanks to the following auxiliary concepts. Two instructions in the same routine are "*siblings*" if their closest enclosing Compounds are the same or nested within one another. (In *i*1; **if** *c* **then** *i*2 *i*3 **else** *i*4 **end** *i*5, all instructions are siblings except for the pairs *i*2, *i*4 and *i*3, *i*4.) In addition, a routine appearing in a loop is its own sibling. The "*dependents*" of a routine *r*, or a call to that routine, are *r* itself and all the routines that it may call directly or indirectly. A qualified call *x*.*r* (…) is "*separate*"if its target *x* is separate. Then two instructions *i*1 and *i*2 are simultaneous if *i*1 is in a dependent of a separate call *c*, and *i*2 is in a dependent of a sibling of *c*. For example in

```
i1
sep.r (…)                -- Where sep is separate
i2
x.s (…)                  -- Where x may be separate or not
```

both *i*1 and *i*2 are simultaneous with all the instructions of *r*, *s* and their dependents.

Now the hold and wait sets. For an entity *x* in the program (formal argument, local variable, attribute) let <*x*> be the handler of the object attached to *x*. The notation generalizes to lists: <*l*> is the set of handlers of all the elements of a list *l*. Let **Current** denote the current object ("this"). Then:

For a call *c* with separate actual arguments *actuals*, appearing in a routine with separate formal arguments *formals*:

- *H* (*c*) = <**Current**> ∪ <*Formals*>
- *W* (*c*) = <*Actuals*>

The call *c* (like any instruction in the same routine) holds the handler of the current object and the handlers of all the formal arguments of the enclosing routine. To proceed, *c* requires exclusive access to the handlers of actual arguments. This property applies both to a synchronous call, such as an unqualified call *r* (*Actuals*), for which the execution will not proceed until it has executed the body of *r*, and to an asynchronous call of the form *sep*.*r* (*Actuals*) for which the execution does not need to wait for *r* to complete or even to start, but does need to obtain exclusive access to the arguments.

The *W* rule ignores lock passing. To avoid the loss of generality, we may fictitiously extend the formal argument list of a routine with the separate arguments of its callers, although a more direct technique is desirable. The example discussed below does not involve lock passing.

To apply these rules, we need a way to determine the handler <*s*> of any separate entity *s*. More precisely, since the goal is to determine whether intersections of processor sets are empty, we need to determine whether <*s*> and <*t*> can be the same for two entities. This will be the case if *t* is aliased to *s*, or to *s*.*x* where *x* is a non-separate field of *s* (so that *s*.*x* has the same handler as *s*). This observation highlights *alias analysis* as the core task of deadlock analysis in an object-oriented concurrency framework.

Previous work [5] [2] has introduced an **alias calculus**. The calculus is a set of rules for computing statically, at any program point in the context of an object-oriented language with references, the *alias relation*: the set of pairs of expressions that may, be aliased when execution reaches that point. (Two expressions denoting references are aliased if they are attached to the same object.)

The rules of the alias calculus give, for every kind *c* of construct in the programming language and any alias relation *ar*, the value of *ar* » *c*: the alias relation that will hold after execution of *c* if *r* held before. For example, *ar* » (*c1* ; *c2*) = (*ar* » *c1*) » *c2*). The reader may consult [2] for the full list of rules. (As may be expected, the alias relation computed by these rules is usually an over-approximation of the aliasings that may actually exist at execution time.) The present discussion assumes that we do have the alias calculus at our disposal. For the purpose of alias analysis, we add *s*.*x* to the aliases of *s* for all non-separate *x*.

In this framework, the above rule for computing the hold and wait sets becomes:

For a call *c* with separate actual arguments *actuals*, appearing in a routine with separate formal arguments *formals*:

- *H* (*c*) = *aliases* ({**Current**} ∪ *Formals*)
- *W* (*c*) = *aliases* (*Actuals*)

where *aliases* (*e*) is the set of expressions possibly aliased to *e* at the given program position. In this formulation, the *H* and *W* sets contain program expressions rather than the actual processors; the expressions act as proxies for these processors. Such an abstraction is necessary in any case since the processors only become known at execution time.

Deadlock analysis then reduces to the following steps.

Deadlock analysis strategy:

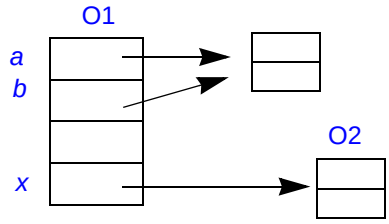
- Determine all pairs [*c1*, *c2*] of simultaneous calls with separate arguments.
- For all calls *c* involved, determine *H* (*c*) and *W* (*c*) according to the rule above, applying aliasing as necessary.
- Determine if all *H* (*c1*) ∩ *W* (*c2*) are non-empty.

7 Example application

The following scenario illustrates deadlock analysis on the two-dining-philosophers example presented earlier.

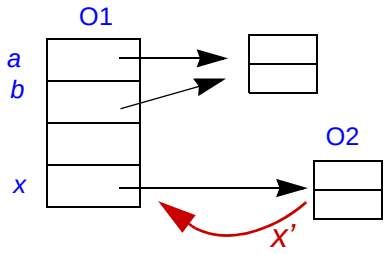
The alias relations at each stage, resulting from the alias calculus, are assumed to come from automatic alias analysis.

In expressing these relations, we need the concept of *negative variable*, introduced in [6] to handle the changes of coordinates that characterize object-oriented programming. To understand this notion assume that *a* is aliased to *b*:



The alias relation is, in the notation of [5], $\overline{a, b}$, meaning that it contains the pair [*a*, *b*] and the symmetric pair [*b*, *a*] (an alias relation is symmetric). Also, *a* and *b* are their own aliases, but for economy we never explicitly include pairs [*x*, *x*] in an alias relation, keeping it irreflexive by convention.

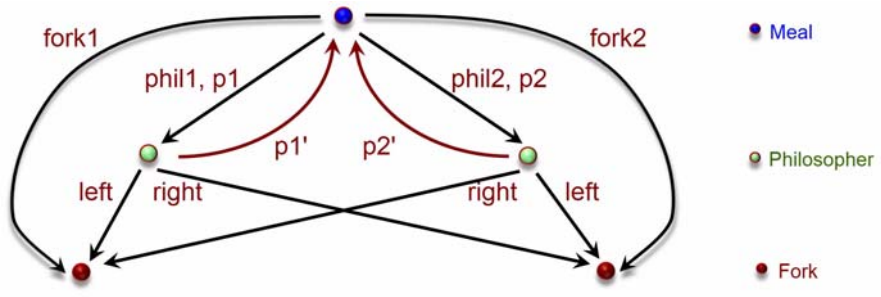
At this point the program executes a qualified call *x*.*r* (…). The routine *r* may change the aliasing situation; but to determine these changes we cannot apply the alias calculus rules to the original relation $\overline{a, b}$ since *a* and *b* are fields of the original object O1 and mean nothing for the target object O2. The relation of interest is *x'*.($\overline{a, b}$), equal by distributivity to $\overline{x'.a, x'.b}$. Here *x'*, the "negation" of *x*, represents a back-reference to the caller as illustrated below. This reference need not exist in the implementation but is necessary for the alias computation. If *ar* is the alias relation obtained by the alias calculus at the end of the body of *ar*, then the result for the caller is *x*.*ar*, where any *x'* will cancel itself out with *x* since *x*.*x'* = **Current** and **Current**.*e* = *e* for any *e*.



Let us now consider the deadlock analysis of the preceding section to the two versions of the dining philosopher program. In the first version, the relevant simultaneous pair is the call *eat* (*left*, *right*), paired with itself (since it is in a loop), in the routine *live*. Prior to aliasing, the hold and wait sets, seen from the philosopher object, are:

- *H* = {**Current**}
- *W* = {*left*, *right*}

Considering both calls to *live*, the reference structure is the following:



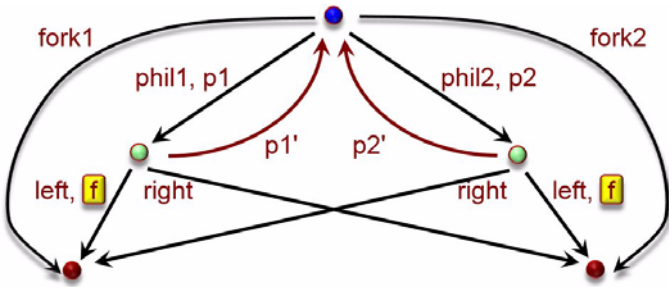
The top node is the root object, of type *MEAL*; the middle nodes are philosopher objects; the bottom nodes are fork objects. Through alias completion we get:

- *H* = {**Current**, *p1'.phil1*, *p1'.p1*}
- *W* = {*left*, *right*, *p1'.fork1*, *p2'.fork2*}

The intersection of these sets is empty: no deadlock. Now consider the version using *eat_bad* (*left*) instead of *eat* (*left*, *right*). The sets at the point of the call to *pick_second* in *eat_bad* are, at the *PHILOSOPHER* level and prior to alias completion:

- *H* = {**Current**, *f* }
- *W* = {*left*, *right*}

The reference structure is the same as above, plus aliasing of *f* to *left*:



Alias completion yields:

- *H* = {**Current**, , *p1'.phil1*, *p1'.p1*, *f* , *left*, *p1'.fork1*, *p2'.fork2*}
- *W* = {*left*, *right*, *p1'.fork1*, *p2'.fork2*, *f* }

Since *W* now includes *f*, aliased to *left*, the intersection is not empty, revealing the possibility of a deadlock.

8 Conclusion and perspectives

The limitations of this work are clear: the technique is not modular; it has not been proved sound; its precision (avoidance of false alarms) is unknown; and it is not yet implemented.

The approach, however, seems promising. The reliance on aliasing seems to open the way for a realistic approach to static deadlock detection, applicable to modern object-oriented programs regardless of the complexity of their run-time object and reference structure.

The next step is to remedy the current limitations and make the approach fully applicable in a practical verification environment [7]. The goal is worth the effort: unleashing the full power of concurrent programming by removing an obstacle that, for decades, has been a nightmare.

Acknowledgments

The research reported here is part of the Concurrency Made Easy project at ETH, an Advanced Investigator Grant of the European Research Council (ERC grant agreement no. 29138). I am grateful to members of the CME project, particularly Scott West, Benjamin Morandi and Sebastian Nanz, for numerous comments on the research. Alexander Kogtenkov and Sergey Vedler were instrumental in the development of the alias calculus. Victorien Elvinger spotted an error in an earlier version.

Bibliography

[1] EVE page (Eiffel Verification Environment) at se.inf.ethz.ch/research/eve/.

[2] Alexander Kogtenkov, Bertrand Meyer and Sergey Velder: *Alias Calculus, Frame Calculus and Frame Inference*, in *Science of Computer Programming*, vol. 97, part 1, January 2015, pages 163-172.

[3] Bertrand Meyer: *Systematic Concurrent Object-Oriented Programming*, in *Communications of the ACM*, vol. 36, no. 9, September 1993, pp. 56-80.

[4] Bertrand Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997 (chapter 32 includes a description of SCOOP).

[5] Bertrand Meyer: *Steps Towards a Theory and Calculus of Aliasing*, in *International Journal of Software and Informatics*, 2011, pages 77-116.

[6] Bertrand Meyer and Alexander Kogtenkov: *Negative Variables and the Essence of Object-Oriented Programming*, in *Specification, Algebra, and Software*, Kanazawa (Japan), 14-16 April 2014, eds. Shusaku Iida, Jose Meseguer and Kazuhiro Ogata, Springer Lecture Notes in Computer Science 8313, pages 171-187, 2014.

[7] Benjamin Morandi, Mischael Schill, Sebastian Nanz and Bertrand Meyer. *Prototyping a concurrency model*, in Int. Conf. on App. of Concurrency to Syst. Design, pages 177-186, available at se.inf.ethz.ch/people/morandi/publications/prototyping.pdf.

[8] Piotr Nienaltowski: *Practical framework for contract-based concurrent object-oriented programming*, PhD thesis, ETH Zurich, 2007, available at se.inf.ethz.ch/old/people/nienaltowski/papers/thesis.pdf.

[9] SCOOP page at cme.ethz.ch.