A Modular Scheme for Deadlock Prevention in an Object-Oriented Programming Model

Scott West, Sebastian Nanz, and Bertrand Meyer

ETH Zurich
firstname.lastname@inf.ethz.ch

Abstract. Despite the advancements of concurrency theory in the past decades, practical concurrent programming has remained a challenging activity. Fundamental problems such as data races and deadlocks still persist for programmers since available detection and prevention tools are unsound or have otherwise not been well adopted. In an alternative approach, programming models that exclude certain classes of errors by design can address concurrency problems at a language level. In this paper we review SCOOP, an existing race-free programming model for concurrent object-oriented programming, and extend it with a scheme for deadlock prevention based on locking orders. The scheme facilitates modular reasoning about deadlocks by associating annotations with the interfaces of routines. We prove deadlock-freedom of well-formed programs using a rigorous formalization of the locking semantics of the programming model. The scheme has been implemented and we demonstrate its usefulness by applying it to the example of a simple web server.

1 Introduction

Concurrent programming has remained a difficult task even for expert programmers, in spite of steady progress in the theory of concurrency. One possible reason is that concurrency is typically added to a language as a secondary concern, via thread libraries. These offer little support for a structured use of synchronization primitives, making it difficult for programmers to reason about their programs. Concurrency research has provided a set of tools, e.g. [13, 18, 6, 3], for addressing the data races and deadlocks that arise from incorrect use of synchronization, but these do not tackle the source of the problem.

Another line of research therefore attempts to create languages that raise the level of abstraction for expressing concurrency and synchronization and hence to make programmers produce better code. Resulting programming models can also exclude certain classes of errors by construction, for example data races [8, 1], usually accepting a penalty in performance or programming flexibility for the sake of program correctness.

One such programming model is SCOOP (Simple Concurrent Object-Oriented Programming) [15, 16]. The model allows objects to be declared of a special type indicated by the keyword **separate**. Calls to routines (methods) on such separate objects are executed asynchronously, i.e. they will be spawned off to a thread

```
class DEADLOCK
create set
feature
  x, y : separate S

  f
    do g (x) end

  g (a : separate S)
    do h (y) end

  h (b : separate S)
    do end

  set (a_x, a_y : separate S)
    do
      x := a_x
      y := a_y
    end
end
```

```
class MAIN
feature
  x, y : separate S

  run (d1, d2 : separate DEADLOCK)
    do d1.f; d2.f end

  make
    local
      d1, d2 : separate DEADLOCK
    do
      create x
      create y
      create d1.set (x, y)
      create d2.set (y, x)

      run (d1, d2)
    end
end
```

Program 1: SCOOP deadlock example

separate from the current one. SCOOP then preserves freedom from object-level data races by requiring that separate objects be *controlled* before features can be invoked on them. Control is obtained by having such an object passed as an argument to a routine: within the scope of the routine, all separate objects that are its actual arguments are automatically locked. In Program 1, `x` is locked by the call to `g` in the body of `f`.

This locking behavior simplifies reasoning about concurrent programs for the programmer, as groups of concurrent objects are protected within the body of a routine and thus "sequential thinking" can be applied in this context. On the other hand, SCOOP offers no protection against deadlocks, a flaw shared with practically all concurrent programming languages.

In this paper, we extend SCOOP with a scheme for deadlock prevention, addressing a critical open problem of this programming model (indeed, Program 1 may deadlock). The scheme is based on establishing an order in which resources can be locked, hence preventing the formation of cyclical locking patterns. As the structure of locking in SCOOP is reflected in the call stack, annotations indicating the locking order are associated with the interfaces of routines, providing modularity at the routine level. We formalize the locking behavior of SCOOP using a structural operational semantics, providing the basis for the deadlock-freedom proof. We provide a technique for statically checking that programs are well-formed according to a well-formedness predicate, and prove that well-formed programs never deadlock. The technique has been implemented and applied to a simple web server programmed in SCOOP.

Other work in this area such as deadlock freedom for active objects in Java [11] provides less versatile structures (trees vs. orders). Techniques of similar power [3], however, are not grounded in an underlying language that is designed to make concurrent programming easier. Lastly, other partial operational semantics [17] only consider liveness properties in the light of model checking.

The remainder of this paper is structured as follows. In Section 2 we give an overview of SCOOP and reason on how deadlock can be detected both dynamically and statically. Section 3 provides a formalization of SCOOP's locking semantics. In Section 4 we describe our deadlock prevention scheme and prove that well-formed programs cannot deadlock. We describe related work in Section 5 and conclude in Section 6.

2 SCOOP programs and their locking semantics

SCOOP [15, 16] is a programming model for concurrency, which can be implemented on top of any object-oriented language. Implementations are currently available for Eiffel [16, 9] (the syntax we use in this paper) and Java [19]. In this section we first provide a short overview of the model, give a description of how deadlock may be identified, and finally show an annotation language for establishing a locking order among resources.

2.1 Overview of the model

Asynchronous calls. The central idea of SCOOP is that every object is associated for its lifetime with a *processor*, an abstract notion denoting a site for computation: just as threads may be assigned to cores on a multi-core system, processors may be assigned to cores, or even to more remote processing units. The (unique) processor associated with a certain object is called its *handler*. Processors may handle multiple objects. A processor can be identified using its *processor tag*.

Processors are an abstraction, allowing the model to be mapped to multithreaded systems, distributed systems, or other concurrent architectures alike. For example in multithreaded systems every processor simply corresponds to one thread on the system, and the processor tag is the thread identifier. Whenever a new processor is created, a new thread is spawned.

Calls on an object are only executed by its handler. For example, if a processor p encounters a call `x.f`, and the object attached to `x` is handled by a processor q then p asks q to evaluate `x.f` on its behalf. If `x.f` does not return a result, processor p can continue executing concurrently with the computation taking place at q. If `x.f` returns a result, the runtime system makes sure that p waits for q to return the result before proceeding.

Type annotations. To make it clear for programmers which calls are executed asynchronously (invoked on objects residing on separate processors) and which calls are synchronous (invoked on objects residing on the current processor), the type system of SCOOP provides a special type indicated by the keyword **`separate`**: if a variable `x` is declared of separate type

```
x : separate X
```

then at creation of `x` with the statement

```
create x
```

a new processor p is created in addition to an object o of type X, and the handler of o is set to p. The type system also allows that the processor tags can be explicitly specified as in

```
x : separate <p> X
y : separate <p> Y
```

which at creation time would place objects x and y on the same processor p. These processor annotations have the scope of a class if applied to attributes of the class, and of the routine's body if applied to local variables of a routine.

Locking behavior. In order to prevent object-level data races in SCOOP, processors that are needed for the execution of a routine are automatically locked by the runtime system before entering the body of the routine; the locks are released upon the completion of the execution of the body. Thus all handlers of separate objects that occur in the body need be locked. The model prescribes that these separate objects need to be *controlled* (passed as arguments to the routine). At routine invocation the runtime system tries to lock the separate arguments' handlers: if the locking succeeds, the execution proceeds into the body of the routine; if it fails because one or more of the handlers are locked by other processors, the runtime system schedules the call to be retried later. In Program 1 the body of the feature f contains the command g(x), the locking behaviour described above would be seen here, as this call is invoked, requesting and locking the object x.

2.2 Deadlock in SCOOP

Knowing how locks and requests appear in the SCOOP model, we can now describe how a deadlock state may be detected. A deadlock state, based on waiting for resource availability as in [5], can be identified

- dynamically: construct a "waits-for" relation; if an element is related to itself in the transitive closure of such a relation, then the system is in a deadlock state. In the setting of SCOOP, the "waits-for" relation contains an association between between processors p and q iff some other processor has a lock on p and is requesting q.
- statically (conservative): arrange the processor tags into a partial order. When the text of the program indicates a lock is taken, verify that it is less than all the other locks that could have been taken at this point. The program text may require some annotations establishing which locks have already been taken.

These two schemes can be applied to Program 1. Reasoning using the dynamic scheme, we see that an instance of class DEADLOCK will lock its attributes x and y in some order when its routine f is called. In class MAIN, two instances d1 and d2 of DEADLOCK are initialized with two separate objects x and y, however their order is reversed between the two instances. By executing `run`, the routine calls `d1.f` and `d2.f` are executed asynchronously, according to the semantics of calls on separate objects `d1` and `d2` outlined above.

As a result of executing `d1.f`, the call `g(x)` is invoked. As `x` is an argument to the routine `g`, the runtime locks `x` for the duration of the call, as prescribed by the semantics for controlled objects outlined above. In particular, `x` will still be locked when the call `h(y)` is invoked, requesting a lock on `y`. The concurrent execution of `d2.f` has an analogous locking behavior, but since `d1` and `d2` have opposite views of `x` and `y`, the locking order is reversed. Hence the calls may ultimately form a cyclical locking pattern, resulting in a deadlock.

To reason statically about the same sequence of calls, one notices that the order of calls can be conservatively approximated by examining the program text, and observing which routines subsequently call other routines. In the case of Program 1, we always know that calling the feature `f` will (for a general routine, may) require that the processor of `x` is locked, followed by `y`. This information can be used statically at the call sites of `d1.f` and `d2.f` to determine that their concurrent execution could lead to a deadlock state.

We have chosen to develop a static technique, as we believe that static techniques encourage the active construction of correct programs, whereas dynamic techniques cater more to a reactive development style.

3 A formal model of SCOOP locking

Our approach uses a static detection scheme, requiring that the interfaces of a program be annotated. This includes routines and types of variables, where the annotations for variables follow the SCOOP-style very closely.

3.1 Annotation language

At the class level, annotations of the following form are allowed:

$$\mathit{class\_header} ::= \textbf{class}\ \mathit{ident}\ \Big|\ \textbf{class}\ \mathit{ident}\ {<}p(,p)^*{>}$$

A class can thus be parameterized with the processor tags it is using. Consider Program 2 for example, an entity `d` based on class `DEADLOCK` uses processors with tags `p` and `q` for the roles of `xp` and `yp`. An instance is declared as follows:

```
d : DEADLOCK <p, q>
```

The preconditions of routines represent the required orderings of processors, expressed using the following syntax (note that we replace the non-strict ordering symbol by a strict ordering symbol in the program text to make it easier to type, however the interpretation should remain non-strict in all cases)

$$\mathit{req} ::= \epsilon\ \Big|\ \textbf{require}\ p < p(, p < p)^*$$

```
   class DEADLOCK <xp,yp>

   feature
     x : separate <xp> S
5    y : separate <yp> S

     f
       require yp < xp
       lock xp
10     do
         g (x)
       end

15   g (a : separate <xp> S)
       require yp < xp
       do h (y) end

     h (b : separate <yp> S)
20     do end

     set (a_x : separate <xp> S;
            a_y : separate <yp> S)
       do
25       x := a_x
         y := a_y
       end
   end
```

Program 2: Annotated DEADLOCK class

For example, in line 16 the routine g is annotated to express that the processor yp, which will be locked as a result of the execution of the body of g, is below processor xp, which is locked by calling g.

In the interface of a routine we state the set of locks that may be taken temporarily during the execution of the routine body.

$$ens ::= \epsilon \;\Big|\; \textbf{lock}\; p(, p)^*$$

For example; in line 9 we state that a lock on x's processor xp may be taken by executing the body of f, as the call g(x) will lock this processor. Note that small changes to the program (re-nesting function calls, for instance) may require the ordering specifications to be modified accordingly.

In Program 2 we require that yp < xp in feature g. Due to the construction of the two deadlock variables d1 and d2 in Program 1, we know that the two classes are instantiated with conflicting requirements: one requires that yp < xp and the other will then necessitate xp < yp. Since these cannot be mutually satisfied, it is impossible to annotate MAIN from Program 1 such that it can satisfy the well-formedness predicate.

3.2 SCOOP program model

Complementing the program annotations, we provide a formalization of SCOOP programs based on the computational model described in Section 2.1. We focus on routines as the basic units of programs, as it is at routine invocation that locks are taken, and at routine return where lock reservations are given up. We disregard classes and class-level processor annotations as they introduce unnecessary complexity in the representation; the formalization could however be extended to include them.

Assume to have a set of (routine) names *Name*. We consider a *program* $\mathcal{P}$ to be a mapping of names to routines

$$\mathcal{P} \in Program = Name \rightarrow Routine$$

where an (unnamed) routine *rtn* is of the form

$$rtn \in Routine = \wp(Tag \times Tag) \times Tag^* \times \wp(Tag) \times Tag \times Expr$$

and we refer to its components using the following notation:

$$rtn = (rtn_{\leq}, rtn_{\text{args}}, rtn_{\text{locks}}, rtn_{\text{res}}, rtn_{body})$$

The component $rtn_{\leq}$, corresponding to programmer provided **`require`** annotations as in Section 3.1, is a relation on processor tags, describing the partial order on processors required by the routine. The component rtn_{args} is the sequence of formal arguments of the routine. The set of locks that may be taken as the result of executing the body of the routine is given by rtn_{locks}, corresponding to **`lock`**; this is the other programmer-provided annotation. The component rtn_{res} specifies whether the routine returns a result or not, and rtn_{body} is the body of the routine (an expression).

An expression e is constructed from the following syntax

$$e \ ::= \ [p] \ \big| \ \mathsf{skip} \ \big| \ \mathsf{create}(p) \ \big| \ e \cdot f(\tilde{e}) \ \big| \ e; e \ \big| \ \mathsf{waitfor}(p) \ \big| \ \mathsf{unlock}$$

where $p \in Tag$ is a processor tag and $f \in Name$ is a name. We write $\tilde{e}$ to abbreviate a sequence of expressions $e_1, \ldots, e_n$, and similarly $\tilde{p}$ for a sequence of processors. We sometimes treat these sequences as sets, i.e. $\tilde{e} = \bigcup_{i=1,\ldots,n}\{e_i\}$. We assume that processor tags can be inferred from an expression e using a mapping $tag_{\mathcal{P}} : Expr \to Tag$; we use $tags_{\mathcal{P}}$ on sequences of expressions.

The syntax elements have the following intuitive meaning. A value on a processor p is represented as $[p]$, abstracting away the actual value. Since the actual value is discarded, assignments in a program text are transformed into only the right-hand side, as it may contain some call (and thus locking). If in a sequence $\tilde{e} = e_1, \ldots, e_n$ all expressions are fully evaluated (i.e. $e_i = [p_i]$ for $i = 1, \ldots, n$), we use the notation $[\tilde{e}]$. The expression skip has no effect. A new processor with tag p is created using $\mathsf{create}(p)$. Calling a routine f on a target t, with a list of arguments $\tilde{a}$ is represented by $t \cdot f(\tilde{a})$. Sequencing of expressions is written as $e_1; e_2$. The remaining two syntax elements $\mathsf{waitfor}(p)$ and unlock do not represent program syntax, but are required for the purpose of modeling the waiting and locking behavior of the runtime system. Waiting on a processor with tag p is expressed as $\mathsf{waitfor}(p)$. The expression unlock represents unlocking of the set of processors that has been taken as a result of the matching routine call.

3.3 Locking semantics

Given the formalization of SCOOP programs, we can proceed to formally defining the part of the program semantics that is critical for reasoning about deadlock. Rather than enabling us to reason about full program correctness, the following rewrite rules embody the behavior of requesting, taking and releasing locks in a SCOOP program.

At runtime, a program $\mathcal{P}$ gives rise to a *process* P which is described by the following syntax:

$$P \ ::= \ p :: e \ \Big| \ P \mid P$$

A process is therefore either an expression e located at a processor with tag p, or a parallel composition of processes. The idea is that a program starts with the initial call f_0 on an initial processor p_0 as $p_0 :: f_0$, and will give rise to more parallel threads (as the result of create) as execution proceeds. A structural equivalence $\equiv$ over processes specifies the commutativity and associativity of the $\mid$ operator; the formal definition of $\equiv$ is standard and omitted from this presentation. We assume that processor tags are unique within processes, i.e. there cannot be a process $P \equiv p :: e \mid q :: e' \mid Q$ such that $p = q$. This property is preserved by process creation.

Processes are operating on a state representing locks and requests only. Formally, we define a *lock state* L as a pair of mappings (L_l, L_r) of the following type:

$$L \in \mathit{LockState} = (\mathit{Tag} \to (\wp(\mathit{Tag}))^*) \times (\mathit{Tag} \to \wp(\mathit{Tag}))$$

Here, L_l is a mapping from a processor (tag) to a stack of sets of processors, representing the processors it currently locks. Although a set of locks would suffice here, having a stack of sets allows for a greater correspondence with the intuition that lock-taking in SCOOP closely follows the call-stack. We define the domain of L as the union of the domains of its components, $\mathit{dom}(L) = \mathit{dom}(L_l) \cup \mathit{dom}(L_r)$. We use the notation $L_l[p \mapsto T]$ for updates, such that the resulting mapping returns T at point p of its domain and is unchanged otherwise. We write $T : \mathit{lcks}$ for a stack obtained by pushing a set of processor tags T on a stack lcks. We write $\bigcup \mathit{lcks}$ for flattening the stack into one set, i.e. if $\mathit{lcks} = T_1, \ldots, T_n$ then $\bigcup \mathit{lcks} = \bigcup_{i=1,\ldots,n} T_i$. L_r is a mapping from a processor to the set of processors it requests locks for. The requested processors are tracked to align our model with the Coffman treatment of when deadlock occurs.

The locking semantics specifies rewrite rules over processes and lock states in the style of a structural operational semantics with transitions of the form:

$$\mathcal{P} \vdash (P, L) \to (P', L')$$

This means that, given a program $\mathcal{P}$ which provides meaning to names of routines occurring in processes, the process P evolves in one step to P' and transforms locking state L to L'.

With this information, we can now look to the rules contained in Table 1 for the definition of the locking semantics. The creation of a new processor q by a processor p gives rise to a new parallel process located at q. If the processor already exists, then this has no effect. These behaviours can be see in the $\textsc{create}_1$ and $\textsc{create}_2$ rules.

The rule $\textsc{seq}$ allows one step to be performed on the left side of a sequential composition, and $\textsc{skip}$ carries its intuitive meaning. For routine target and argument evaluation: $\textsc{eval-trg}$ and $\textsc{eval-arg}$ enforce that targets are fully evaluated before arguments are evaluated. In $\textsc{eval-arg}$, the arrow $\twoheadrightarrow$ represents

$$\textsc{create}_1\quad\frac{p \neq q \qquad Q \not\equiv q :: e \mid Q'}{\begin{array}{l}\mathcal{P} \vdash (p :: \mathsf{create}(q) \mid Q, L) \rightarrow \\ \qquad (p :: \mathsf{skip} \mid q :: \mathsf{skip} \mid Q, L)\end{array}}$$

$$\textsc{create}_2\quad\frac{p = q \ \vee\ Q \equiv q :: e \mid Q'}{\begin{array}{l}\mathcal{P} \vdash (p :: \mathsf{create}(q) \mid Q, L) \rightarrow \\ \qquad (p :: \mathsf{skip} \mid Q, L)\end{array}}$$

$$\textsc{eval-trg}\quad\frac{\begin{array}{c}\mathcal{P} \vdash (p :: t \mid Q, L) \rightarrow \\ (p :: t' \mid Q', L')\end{array}}{\begin{array}{l}\mathcal{P} \vdash (p :: t \cdot f(\tilde{a}) \mid Q, L) \rightarrow \\ \qquad (p :: t' \cdot f(\tilde{a}) \mid Q', L')\end{array}}$$

$$\textsc{eval-arg}\quad\frac{\begin{array}{c}\mathcal{P} \vdash (p :: \tilde{a} \mid Q, L) \twoheadrightarrow \\ (p :: \tilde{a}' \mid Q', L')\end{array}}{\begin{array}{l}\mathcal{P} \vdash (p :: [q] \cdot f(\tilde{a}) \mid Q, L) \rightarrow \\ \qquad (p :: [q] \cdot f(\tilde{a}') \mid Q', L')\end{array}}$$

$$\textsc{skip}\quad\frac{}{\begin{array}{l}\mathcal{P} \vdash (p :: \mathsf{skip}; e \mid Q, L) \rightarrow \\ \qquad (p :: e \mid Q, L)\end{array}}$$

$$\textsc{seq}\quad\frac{\begin{array}{c}\mathcal{P} \vdash (p :: e_1 \mid Q, L) \rightarrow \\ (p :: e_1' \mid Q', L')\end{array}}{\begin{array}{l}\mathcal{P} \vdash (p :: e_1; e_2 \mid Q, L) \rightarrow \\ \qquad (p :: e_1'; e_2 \mid Q', L')\end{array}}$$

$$\textsc{equiv}\quad\frac{\begin{array}{c}P \equiv Q \qquad Q' \equiv P' \\ \mathcal{P} \vdash (Q, L) \rightarrow (Q', L')\end{array}}{\begin{array}{c}\mathcal{P} \vdash (P, L) \rightarrow \\ (P', L')\end{array}}$$

$$\textsc{call-nores}\quad\frac{\begin{array}{c}\mathcal{P}(f)_{result} = \text{None} \\ Q \equiv q :: e \mid Q' \qquad q \in \bigcup L_l(p)\end{array}}{\begin{array}{l}\mathcal{P} \vdash (p :: [q] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow \\ \qquad (p :: \mathsf{skip} \mid q :: e; [q] \cdot f([\tilde{a}]) \mid Q', L)\end{array}}$$

$$\textsc{call-res}\quad\frac{\begin{array}{c}\mathcal{P}(f)_{result} = \text{Some}(v) \\ Q \equiv q :: e \mid Q' \qquad q \in \bigcup L_l(p)\end{array}}{\begin{array}{l}\mathcal{P} \vdash (p :: [q] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow \\ \qquad (p :: \mathsf{waitfor}(q) \mid q :: e; [q] \cdot f([\tilde{a}]) \mid Q', L)\end{array}}$$

$$\textsc{req-lck}\quad\frac{\begin{array}{c}need = \tilde{a} - (\bigcup L_l(p) \cup \{p\}) \\ L_r' = L_r[p \mapsto need] \qquad L_l' = L_l\end{array}}{\begin{array}{l}\mathcal{P} \vdash (p :: [p] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow \\ \qquad (p :: [p] \cdot f([\tilde{a}]) \mid Q, L')\end{array}}$$

$$\textsc{lock}\quad\frac{\begin{array}{c}L_r' = L_r[p \mapsto \emptyset] \qquad \tilde{a}' = \mathcal{P}(f)_{arg} \\ \left(\bigcup_{x \in \mathrm{dom}(L)} \bigcup L_l(x)\right) \cap L_r(p) = \emptyset \\ L_l' = L_l[p \mapsto L_r(p) : L_l(p)]\end{array}}{\begin{array}{l}\mathcal{P} \vdash (p :: [p] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow \\ \qquad (p :: \mathcal{P}(f)_{body}[\tilde{a}/\tilde{a}']; \mathsf{unlock} \mid Q', L')\end{array}}$$

$$\textsc{ret}\quad\frac{\begin{array}{c}Q \not\equiv q :: \mathsf{waitfor}(p) \mid Q' \qquad L_r' = L_r \\ L_l(p) = T : lcks \qquad L_l' = L_l[p \mapsto lcks]\end{array}}{\begin{array}{c}\mathcal{P} \vdash (p :: [v]; \mathsf{unlock} \mid Q, L) \rightarrow \\ (p :: [v] \mid Q, L')\end{array}}$$

$$\textsc{ret-wait}\quad\frac{\begin{array}{c}Q \equiv q :: \mathsf{waitfor}(p) \mid Q' \qquad L_r' = L_r \\ L_l(p) = T : lcks \qquad L_l' = L_l[p \mapsto lcks]\end{array}}{\begin{array}{c}\mathcal{P} \vdash (p :: [v]; \mathsf{unlock} \mid Q, L) \rightarrow \\ (p :: \mathsf{skip} \mid q :: [v] \mid Q', L')\end{array}}$$

$$\textsc{unlock}\quad\frac{\begin{array}{c}L_l(p) = T : lcks \qquad L_r' = L_r \\ L_l' = L_l[p \mapsto lcks]\end{array}}{\begin{array}{c}\mathcal{P} \vdash (p :: \mathsf{unlock} \mid Q, L) \rightarrow \\ (p :: \mathsf{skip} \mid Q, L')\end{array}}$$

Table 1: SCOOP Rewrite Rules

performing a single rewrite step on a sequence of expressions. To reorder the constituent processes of a program during rewriting, the EQUIV rule is available.

Once the target and arguments of a call are both fully evaluated, the call can be invoked. In the case where the call has no result, CALL-NORES moves the call to the target processor, to be executed after the current tasks of the target processor; the caller proceeds without waiting. Recall that we use the notation $[\tilde{a}]$ to describe a fully evaluated sequence of expressions, and $\bigcup L_l(p)$ for flattening the stack of locks $L_l(p)$. To make a call on a separate target, we require the processor p to hold a lock on the target processor q with the condition $q \in \bigcup L_l(p)$. When the call has a result, the dispatching processor must wait on the result from the target processor, as in CALL-RES.

Upon a call arriving on its target processor, the required locks must be requested, specified in REQ-LCK. We only request the locks we do not already hold, which are collected in the set *need*; the local processor is never *need*ed. Once the requests have been made, they are transferred to the lock set (LOCK) of the processor when no other processor has any of the locks. Then the body of the routine is scheduled for evaluation, followed by a request to unlock all initially requested locks after the execution of the body has been completed. Here we use the notation $\mathcal{P}(f)_{body}[\tilde{a}/\tilde{a}']$ to substitute the sequence of actual arguments $\tilde{a}$ for the formal arguments $\tilde{a}' = \mathcal{P}(f)_{arg}$ within the body $\mathcal{P}(f)_{body}$ of routine f. In the previous two rules, the sequence of values $[\tilde{a}] = [p_1], \ldots, [p_n]$ is reinterpreted in set computations as a set of processors, i.e. $\tilde{a} = \bigcup_{i=1,\ldots,n} \{p_i\}$.

The waitfor primitive allows a value that has been computed on a target processor q to be transferred to the processor p that is waiting for it (compare rule CALL-RES). As the returning of the value also completes a call, the locks that have been taken as a result of the call are also released (L_l' is obtained from L_l by popping one element off the stack). Two rules are required to return values to callers (RET and RET-WAIT), one which would be the result of a non-separate call (no waitfor), and one which has an accompanying waitfor on another processor:

For the case where a call has been completed but no result is returned (compare rule CALL-NORES) there may be no value $[v]$ sitting before the unlock, so an analogous rule for unlocking is needed with UNLOCK.

Example 1. To illustrate the use of the rewrite rules, we apply them to Program 1. System execution starts with a call `make` on an initial processor p. We show an execution step of the body of `make`, demonstrating an application of rule CREATE_1 on the instruction **`create`** `x`:

$$(p :: \mathsf{create}(q); e \mid Q, L) \to (p :: e \mid q :: \mathsf{skip} \mid Q, L)$$

Here we assume that the processor tag of the local variable `x` is q and can be obtained with a mapping $\mathit{tag} : \mathit{Name} \to \mathit{Tag}$.

The other **`create`**-statements will give rise to more concurrent processes. Finally, the routine `run` is called, and we assume that $\mathit{tag}(\texttt{d1}) = r_1$ and $\mathit{tag}(\texttt{d2}) = r_2$ to get the following derivation.

$$\begin{array}{l}
(p :: [p] \cdot \texttt{run}([r_1],[r_2]); e' \mid r_2 :: \mathsf{skip} \mid r_1 :: \mathsf{skip} \mid Q', L) \to \\
(p :: [p] \cdot \texttt{run}([r_1],[r_2]); e' \mid Q'', (L_l, L_r[p \mapsto \{r_1, r_2\}])) \to \\
(p :: [r_1] \cdot \texttt{f}; [r_2] \cdot \texttt{f}; \mathsf{unlock} \mid Q'', (L_l[p \mapsto \{r_1, r_2\} : L_l(p)], L_r'[p \mapsto \emptyset])) \to \\
(p :: [r_2] \cdot \texttt{f}; \mathsf{unlock} \mid r_2 :: \mathsf{skip} \mid r_1 :: \mathsf{skip}; [r_1] \cdot \texttt{f} \mid Q', L'')
\end{array}$$

Here, the first step is due to rule REQ-LCK and shows that the processors of `d1` and `d2` are added to the request set of p. The second step is then according to rule TAKE-LCK, and shows that the requested locks (which are available) are taken by pushing them on p's stack of locks. The last step is an application of rule CALL-NORES and shows how an asynchronous call is transferred to its handling processor. Applications of rules SKIP and SEQ are omitted for brevity.

4 Deadlock prevention scheme

In this section we present a scheme for deadlock prevention, based on annotations in Section 3.1. We define well-formedness of annotated programs. We prove that well-formed programs cannot deadlock, based on our formalization of the locking semantics in Section 3.3.

4.1 Well-formed programs

The scheme for ensuring that a program is well-formed ensures that there exists, for each routine, a consistent processor ordering (through $rtn_{\leq}$). Additionally, it ensures that locks are declared (rtn_{locks}) properly, and within the scope of these declared locks the callee's locks (rtn'_{locks} instantiated by its arguments) do not lose any of the knowledge that the declared locks are held. The well-formedness property of a program can be formally stated as a predicate:

$$\mathit{wfProgram}_{\mathcal{P}} = \forall rtn \in \mathit{range}(\mathcal{P}).\ \mathit{wfRoutine}_{\mathcal{P}}(rtn)$$

A well-formed routine must ensure that it's interface is well-formed (first clause) and also that the routine body is consistent with the interface (second clause):

$$\mathit{wfRoutine}_{\mathcal{P}}(rtn) = \mathit{isOrder}(rtn_{\leq}) \wedge \mathit{wfExpr}_{\mathcal{P}}(rtn_{\leq}, rtn_{\mathit{locks}}, rtn_{\mathit{body}})$$

The definition of a well formed expression allows neither waitfor nor unlock in the program text, these are only inserted at runtime by the rewrite process. The well-formedness of expressions is thus given by the following definition:

$$\begin{aligned}
&\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, [p]) &&= \mathit{True}\\
&\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, \mathsf{skip}) &&= \mathit{True}\\
&\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, \mathsf{create}(p)) &&= \mathit{True}\\
&\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, e_1; e_2) &&= \mathit{wfExpr}_{\mathcal{P}}(\leq, lks, e_1) \wedge \mathit{wfExpr}_{\mathcal{P}}(\leq, lks, e_2)\\
&\mathit{wfExpr}_{\mathcal{P}}(\leq, lks, t \cdot f(\tilde{a})) &&= \mathit{inst}_{\leq} \subseteq \leq \wedge\, \mathit{wfLevels}_{\mathcal{P}}(\leq, \mathit{inst}, lks, \tilde{a}) \wedge\\
& &&\quad \forall a \in \tilde{a}.\ \ \mathit{tag}_{\mathcal{P}}(a) \leq \mathit{tag}_{\mathcal{P}}(t) \wedge \mathit{wfExpr}_{\mathcal{P}}(\leq, lks, a)\\
& &&\qquad\quad \text{where } \ \mathit{inst} = \mathcal{P}(f)[\tilde{a}/\mathcal{P}(f)_{\mathrm{args}}]
\end{aligned}$$

We treat the cases of values, skip, create, and sequencing with less detail here: they are either immediately well-formed or are well-formed based on a trivial recursion. The first clause of the call-case of *wfExpr* states that the instantiated routine interface must have its order consistent with the context-order ($\leq$). The second clause states that the lock-level is respected. The third clause states that each argument is a well-formed expression, and its processor is less than the target of the call.

$$\mathit{wfLevels}_{\mathcal{P}}(\leq, \mathit{inst}, lks, \tilde{a}) = ((\mathit{inst}_{\mathit{locks}} \times lks) \ \subseteq \leq) \wedge (\mathit{tags}_{\mathcal{P}}(\tilde{a}) \subseteq \mathit{inst}_{\mathit{locks}})$$

The first clause of *wfLevels* has all associations between the declared locks of the call and the context locks being also in the order relation. In other words, this states that each declared lock of a call must be less than all of the context-locks, so that we only lock "down" the partial order. Since a routine may have no arguments and still lock some processors in its body we compare context-locks against the **lock** clause, and not the arguments. The second clause states that if a routine does have arguments, then these arguments must be a subset of the **lock** clause, for consistency.

Example 2. For the Program 2, we show the evaluation the predicate *wfExpr* on the call of the routine `g` in the body of routine `f`. To make the example more varied, assume that the argument $[xp]$ and the corresponding lock of g are replaced by $[zp]$.

We let $\mathit{ord} = \{(yp, xp), (xp, t), (xp, xp), (yp, yp), (t, t)\}$. As $(xp, t) \in \mathit{ord}$ and $\mathit{inst}_{\leq} = \mathit{ord}$, the predicate is satisfied:

$$\begin{aligned} \mathit{wfExpr}(\mathit{ord}, \{xp\}, [t] \cdot g([xp])) = {} & (\forall a \in [xp].\ (\mathit{tag}_{\mathcal{P}}(a), t) \in \mathit{ord} \wedge \\ & \quad \mathit{wfExpr}_{\mathcal{P}}(\mathit{ord}, \{xp\}, [xp])) \\ & \wedge\ \mathit{inst}_{\leq} \subseteq \mathit{ord} \\ & \wedge\ \mathit{wfLevels}(\{(yp, xp), (xp, t)\}, \mathit{inst}, \{xp\}, \{xp\}) \end{aligned}$$

Here we use that

$$\begin{aligned} \mathcal{P}(g) &= (\{(yp, zp), (zp, t), (zp, zp), (yp, yp), (t, t)\}, [zp], \{zp\}, None) \\ \mathit{inst} &= \mathcal{P}(g)[[xp]/[zp]] = (\mathit{ord}, [xp], \{xp\}, None) \end{aligned}$$

and that the predicate *wfLevels* is satisfied because values are well-formed, and (xp, xp) is in the order (reflexivity).

$$\mathit{wfLevels}(\mathit{ord}, \mathit{inst}, \{xp\}, [xp]) = (\{xp\} \times \{xp\}) \subseteq \mathit{ord} \wedge \mathit{tags}_{\mathcal{P}}([xp]) \subseteq \{xp\}$$

4.2 Deadlock freedom

Intuitively, our scheme ensures that there exists a global ordering for every well-formed program, and also that during execution of this program each processor obeys an order in which to take locks. Deadlock-freedom follows from the fact that the acyclicity of the locking state is preserved under any execution step.

To formalize these ideas, we build on notion of a locking graph from [5]. We do not directly show that the rewriting of the operational semantics can not get "stuck" due to lock requests, although this property follows from the locking graph formalization. Translated to our setting, a locking graph has processors (resources) as nodes. There is an edge (p, q) in the graph if some process has locked processor p while requesting processor q. A locking-state L induces a locking-graph relation $\mathit{graph}(L)$ as follows, where $\mathit{Id}_{\mathit{dom}(L)}$ is the identity relation on processors in the domain of L:

$$\mathit{graph}(L) = \mathit{Id}_{\mathit{dom}(L)} \cup \left(\bigcup_{p \in \mathit{dom}(L)} L_l(p) \times L_r(p) \right)$$

The information provided by the lock state L, and associated locking-graph, is not rich enough to prove the properties that will be needed. We therefore introduce two new concepts: a lock-barrier $L_b : \mathit{Tag} \to (\wp(\mathit{Tag}))^*$ and a runtime ordering $L_{\leq} \in \wp(\mathit{Tag} \times \mathit{Tag})$. The lock barrier represents the set of upper bounds on the locks we are allowed to request. The runtime ordering is the ordering which is built up during execution. For the sake of the proof, the locking semantics has to be instrumented with these concepts. The minimal additions to the semantics are shown in Table 2. For our approach, all locks taken have to stay below the current locking barrier at any time, and the runtime ordering is the order that is built at runtime as a result of the order annotations.

$$\textsc{lock}\quad \frac{L'_b(p) = (f_{locks}[\tilde{a}/\tilde{a}']) : L_b(p) \quad L'_{\leq} = (L_{\leq} \cup (f_{\leq}[\tilde{a}/\tilde{a}']))^* \quad \dots}{\dots} \qquad \textsc{ret}\quad \frac{L_b(p) = b : L'_b(p) \quad \dots}{\dots}$$

$$\textsc{ret-wait}\quad \frac{L_b(p) = b : L'_b(p) \quad \dots}{\dots} \qquad \textsc{unlock}\quad \frac{L_b(p) = b : L'_b(p) \quad \dots}{\dots}$$

Table 2: Instrumented rules

We prove that the following predicate, *sound*, is invariant under execution. The predicate states that the runtime ordering $L_{\leq}$ is indeed a partial order, that the locking barrier is respected, and that the locking graph is acyclic.

$$\begin{aligned} \mathit{sound}(L) = {} & \mathit{isOrder}(L_{\leq}) & (1) \\ & \wedge\ \forall p \in \mathit{dom}(L).\ \ \mathit{top}(L_b(p)) \times \bigcup L_l(p) \subseteq L_{\leq} & (2) \\ & \wedge\ \mathit{graph}(L) \subseteq L_{\leq}^{-1} & (3) \end{aligned}$$

Here, $L_{\leq}^{-1}$ denotes the converse of the relation $L_{\leq}$, and *top* denotes the first element of a sequence.

Theorem 1. *Given a well-formed program $\mathcal{P}$ and an instrumented rewrite rule $\mathcal{P} \vdash (P, L) \to (P', L')$, sound$(L)$ implies sound(L').*

Briefly, the third clause is of primary concern; if the locking-graph ($\mathit{graph}(L)$) is a subset of an order, then it must be acyclic. Since $L_{\leq}$ is an order, thus acyclic, so is its inverse.

The initial two clauses support this goal, with the first establishing that as the program executes the relation that is specified piece-wise in the routine annotations is indeed an order. This fact follows from the definition of *wfRoutine* and the instantiation of the routines in the first clause of the call-case of *wfExpr*.

The second clause of *sound* states that the new upper-bound on locks is below all other locks that have already been acquired by the processor p. The proof of this property is garnered from the Cartesian product in the *wfLevels* predicate, which imposes that when locks are taken, they are statically less than every lock taken by the surrounding procedure. When function calls are nested, these transitively combine to ensure that locks requested by a processor p are less than all other locks currently held by that processor. Since we know that the locking scheme preserves the order relation, it must also preserve the inverse order relation, which is the essential property desired to prove the third clause.

4.3 Usage and Tool Support

We have implemented the static checking of our scheme in a prototype tool, written in Haskell [9]. Using this tool we successfully verified that a simple web server is deadlock-free, a portion of which can be seen in Program 3.

To reduce the annotation burden, we have also implemented a simple annotation inference algorithm. The annotations shown in Program 2 can be automatically inferred using the tool. The simple inference scheme automatically

```
db : separate <d> DATABASE

req (sock : separate <s> NET_SOCK)
  require d < s
  local
    last     : STRING
    http_req : HTTP_REQUEST
  do
    create http_req.make ()

    from read_line (sock)
    until last.is_equal (cr)
    loop
      http_req.add_field (last)
      read_line (sock)
    end
    update_database (db, http_req)
    process_request (http_req)
  end
```

Program 3: HTTP request processing

identifies **separate** class attributes with processor tags and lifts the tags to the class header. It also propagates **lock** and **require** clauses appropriately, based on calls within the body of a routine. For example, at a call-site, the **require** clause of the call would be automatically appended to the containing routine's **require** clause; a similar approach is taken for the **lock** clause. This typically makes the manual annotation burden light.

5 Related work

The problem of describing, detecting, and preventing deadlocks in concurrent systems has spawned research based on a variety of approaches. Necessary conditions for a deadlock to occur have been described in a seminal work by Coffman et al. [5]. *Dynamic techniques* can be used to detect deadlocks, e.g. using techniques such as those presented by Bensalem et al. [2]. The fundamental approach in this work is to instrument the program and use this runtime locking information to detect locking cycles. The benefit is that this technique can be less conservative than our approach, but it is based on actual program traces, and the results are, therefore, not sound.

Static techniques rely on programmer annotations to indicate a partial order among the program's locks, and statically check whether this order is abided by; this general idea is also the basis of our approach. Korty [13] proposed a Lint-like tool for detecting deadlocks in programs with semaphores, however without soundness guarantees. Extended static checking for Modula-3 [6] and Java [7] uses program specifications in the style of Eiffel [15], from which verification conditions are generated and checked with an automatic theorem prover. Warnings are provided for various program errors, including deadlock. Being based on Eiffel-style specifications, annotations in this approach are similar to our scheme. However, no soundness guarantees are given whereas we guarantee deadlock-freedom for well-formed programs. Jacobs et al. [10] also generate verification conditions for annotated programs, and guarantee deadlock-freedom for programs verified with a static checker. In contrast to our work, they use a programming model for Java-like languages which is very different from SCOOP, and do not provide a rigorous formal locking semantics.

A number of static approaches to deadlock prevention are based on *type systems*, in particular using ownership types [4]. Boyapati et al. [3] have introduced the ability, as in our approach, to create a directed acyclic graph, well-order, or tree to represent the underlying partial order. In contrast to this approach, our scheme makes it possible to declare locking orders in a routine-local manner, which allows for a finer-grained modularity.

Our work is distinguished from the above approaches in that it has a higher-level concurrency model, not based on traditional threads, and thus has a coarser-grained locking model.

Using a model similar to SCOOP, Kerfoot et al. [11] use types to ensure deadlock freedom for active objects [14]. Ownership types impose a hierarchy on active objects, but the variety of ownership-structures that are permitted are limited. Only trees are allowed, where our approach can support a general directed acyclic graph. Ostroff et al. [17] develop a partial operational semantics for SCOOP, and consider liveness properties of programs in the context of model checking. While the approach can detect deadlocks, it is not modular, thus does not scale to large programs. Kobayashi [12] gives π-calculus a type system that is able to infer and verify deadlock properties about a program. It gives a versatile approach that is even able to reason about recursive processes. However, our work targets a new model of computation that is more immediately amenable to traditional imperative programming.

6 Conclusion

In this paper we have presented a static technique for deadlock prevention in SCOOP, an object-oriented programming model for concurrency. We found that the model supports well reasoning about deadlock, as lock acquisition and release are related to routine invocation and return. This allows the annotations to be attached to the interface of routines, facilitating modular (per-routine) proofs of correctness. This aspect is essential in practice as it is easier to reason about deadlock when it is assured that local changes will not affect the overall result. An implementation of the scheme is available, and has been successfully applied to the example of a web server written in SCOOP.

Adding a deadlock prevention technique for SCOOP removes a critical deficiency of this particular model, but the results also provide important general lessons learned. While sound and scalable programming models for concurrency are overdue, the divide between formally driven language developments (such as process calculi) and concurrent programming language design still seems to be large. This work showcases how one may bridge this gap by using formal reasoning to derive techniques that can be applied to practical programming languages.

In future work we will investigate the possibility of statically avoiding deadlock by creating some objects on the same processor when not rejected by other constraints, expanding on the annotation inference techniques. Work on the semantic foundations of the programming model provides also many avenues for future research. The distributed nature apparent in the semantics can give important insights into extending the programming model for distribution. Also, variants of the semantics can be studied, for example to provide insights about possible performance improvements.

Acknowledgments. This work is part of the SCOOP project at ETH Zurich, which has benefitted from grants from the Hasler Foundation, the Swiss National Foundation, Microsoft (Multicore award) and ETH (ETHIIRA).

References

1. D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *Proc. OOPSLA'00*, pages 382–400. ACM, 2000.
2. S. Bensalem, J. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *PADTAD '06*, pages 41–50. ACM, 2006.
3. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proc. OOPSLA'02*, pages 211–230. ACM, 2002.
4. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 33(10):48–64, 1998.
5. E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.
6. D. L. Detlefs, R. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq SRC, 1998.
7. C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI'02*, pages 234–245. ACM, 2002.
8. C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
9. SCOOP homepage. http://scoop.origo.ethz.ch/, 2010.
10. B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *Proc. ICFEM'06*, volume 4260 of *LNCS*, pages 420–439. Springer, 2006.
11. E. Kerfoot, S. McKeever, and F. Torshizi. Deadlock freedom through object ownership. In *Proc. IWACO'09*, pages 1–8. ACM, 2009.
12. Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR '06*, pages 233–247. Springer-Verlag, 2006.
13. J. A. Korty. Sema: A lint-like tool for analyzing semaphore usage in a multithreaded UNIX kernel. In *USENIX Winter Technical Conference*, 1989.
14. R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design*, pages 483–499. Addison-Wesley, 1996.
15. B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition, 1997.
16. P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming.* PhD thesis, ETH Zurich, 2007.
17. J. S. Ostroff, F. Torshizi, H. F. Huang, and B. Schoeller. Beyond contracts for concurrency. *Formal Aspects of Computing*, 21(4):319–346, 2009.
18. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
19. F. Torshizi, J. S. Ostroff, R. F. Paige, and M. Chechik. The SCOOP concurrency model in Java-like languages. In *Proc. CPA'09*. IOS Press, 2009.