Refined access control policy for SCOOP

Piotr Nienaltowski

Chair of Software Engineering
Swiss Federal Institute of Technology Zurich
CH-8092 Zurich, Switzerland
piotr.nienaltowski@se.inf.ethz.ch

Abstract. The SCOOP model provides programmers with a simple extension of Eiffel that allows them to produce high-quality concurrent applications with little more effort than sequential ones. The model is simple yet powerful; nevertheless, its access control policy is pessimistic: (1) all separate actual arguments of a feature call are locked, even if it is not necessary, and (2) at most one client object can access a given supplier object at any time. As a result, SCOOP-based programs are deadlock-prone. Additionally, some interesting synchronisation scenarios, e.g. reader-writer, cannot be implemented efficiently. This paper presents an extended access control policy for SCOOP that makes it possible to specify which arguments of a routine call should be locked; it also allows for shared access to supplier objects thanks to the interleaving of pure query calls. Additionally, we introduce a mechanism for lock-passing that increases the expressive power of the model, thus allowing the programmer to write programs that were impossible to implement in the original SCOOP model.

1 Introduction

Controlling access to shared resources is one of the main problems in concurrent programming. In non-object-oriented settings the concept of critical section is used to prevent the dangerous interference of several processes. At most one process may be executing the critical section at any given time. Solutions to conflict problems involve proper synchronisation among processes, so that they have to wait for executing a critical section if another process is accessing the shared resource that the critical section is protecting. This kind of synchronisation is called mutual exclusion. The situation changes significantly when we deal with object-oriented computations. Explicit critical sections are not necessary because they may be encapsulated in class routines, as in the SCOOP model [1]. The most important question is how to ensure that concurrent calls to the routines of the same object do not cause deadlock and do not violate the integrity of the object (i.e. the invariant of its base class). An appropriate locking policy should be applied in order to ensure these two conditions. The SCOOP model proposes such a policy. SCOOP-based applications satisfy the safety requirements – they exhibit no data races and no invariant violations due to parallelism. Unfortunately, this comes at a very high price: all accesses to a
separate supplier object must be wrapped in a routine body that represents a critical section; this results in a very coarse-grained parallelism. The policy also applies to query calls that do not modify the state of the supplier. As a result, certain scenarios, e.g. reader-writer synchronisation, cannot be implemented efficiently. In most cases, the amount of locking is higher than necessary because all separate actual arguments of a feature call must be locked. Such a pessimistic locking policy makes SCOOP-based programs more deadlock-prone.

We present three ways of relaxing the access control policy: (1) we introduce a mechanism for specifying which arguments of a routine call should be locked, and (2) we allow shared locking and interleaving of pure query calls. Finally, we propose (3) a simple lock-passing mechanism that greatly improves the flexibility of SCOOP. We illustrate the discussion with numerous code examples.

The article is organised in the following way. Section 2 shortly introduces the SCOOP model and discusses its locking policy. Section 3 describes the extension that allows for precise specification of formal arguments to be locked. Section 4 presents the shared locking mechanism based on the concept of pure queries. Section 5 introduces the lock-passing mechanism. Section 6 discusses related work. Finally, Section 7 concludes the article and describes future research directions.

Note that the need for lock passing – as a way to clarify SCOOP semantics – was initially pointed out by Phil Brooke and later reflected in the CSP semantics for SCOOP [2] in the form of transitive locking. We follow a different approach here; our goal is to increase the flexibility of the model while preserving the possibility to simulate the original locking semantics. The differences between both solutions are discussed in section 6.

2 SCOOP model

The SCOOP model (Simple Concurrent Object-Oriented Programming) offers a disciplined approach to building high-quality concurrent systems. The idea of SCOOP is to take object-oriented programming as given, in a simple and pure form based on the concepts of Design by Contract [3], which have proved highly successful in improving the quality of sequential programs, and extend them in a minimal way to cover concurrency and distribution. The extension consists of just one keyword separate; the rest of the mechanism largely derives from examining the consequences of the notion of contract in a non-sequential setting. Writing applications with SCOOP is extremely simple, since programmers do not need to deal with low-level concepts typically used in concurrent programming (semaphores, rendezvous, monitors etc.).

2.1 Processors

SCOOP uses the basic scheme of object-oriented computation: the feature call \( x.f(a) \), which should be understood in the following way: the caller object calls feature \( f \) on the supplier object attached to \( x \), with the argument \( a \). In a sequential setting, such calls are synchronous, i.e. the caller is blocked until the
supplier has terminated the execution of the feature. To introduce concurrency, SCOOP allows the use of more than one processor to handle the execution of features. A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. If different processors are used for handling the caller and the supplier objects, the feature call becomes asynchronous: the computation on the caller object can move ahead without waiting for the call to terminate. Processors are the principal concept that SCOOP adds to the sequential object-oriented framework. Contrary to a sequential system, a concurrent system may have any number of processors, independently of the number of available CPUs.

2.2 Separate calls

A declaration of an entity, which normally appears as x: SOME_CLASS may now also be of the form x: separate SOME_CLASS. Keyword separate indicates that entity x is handled by a (potentially) different processor, so that calls on x might be asynchronous and may proceed in parallel with the rest of computation. With such a declaration, x becomes a separate entity. If the target of a call is a separate expression – a separate entity or an expression involving at least one separate entity – such call is referred to as separate call.

2.3 Synchronisation

SCOOP caters for the synchronisation and communication needs of concurrent programming such as mutual exclusion, locking, and waiting by relying on Design by Contract and argument passing.

Mutual exclusion A basic rule of SCOOP says that a separate call an\_x.\ f (a) (where an\_x is separate) is only permitted if an\_x appears as formal argument of the enclosing routine; calling a routine with such a separate argument will make the client object wait until the corresponding separate supplier object is exclusively available to the caller. So, if the client calls r (x), where routine r is defined as

r (an\_x: separate X)
\[\begin{array}{l}
\text{do} \\
\quad an\_x.f (a) \\
\text{end}
\end{array}\]

the call will wait until the processor handling x is available to the client (i.e. no other client is using it). This rule provides the basic synchronisation mechanism for SCOOP. It avoids the most common mistake in concurrent programming that consists in assuming that, when making two successive calls on a separate object, e.g.
```
my_stack.push (some_value)
...
x := my_stack.top
```

nothing may happen to the object represented by `my_stack` between the two calls. In the example, we would expect that the object assigned to `x` is indeed the object denoted by `some_value` that we just pushed on `my_stack`. Unfortunately, such “sequential thinking” does not apply in a concurrent setting, since other clients may interfere with the object referred to by `my_stack` between the two calls. In SCOOP, routine bodies represent critical sections (w.r.t. to their separate arguments) - the client gets an exclusive access to all the processors that handle the separate arguments of the routine. In the example above, `my_stack` must be an argument of the enclosing routine, therefore there is no danger that another client “jumps in” and modifies the state of the supplier object between two consecutive calls issued by our client.

**Condition synchronisation** SCOOP provides the support for condition synchronisation by giving a different semantics to preconditions in a concurrent context. Precondition clauses that involve separate calls become *wait-conditions*; the client object is forced to wait until they are satisfied. We do not discuss the condition synchronisation mechanism any further here because it is not influenced by the new access control policy; interested readers should refer to [1] for more details.

**Resynchronisation** No special mechanism is required for a client object to resynchronise with its supplier after a separate call `x. f (a)` has gone off in parallel. The client will wait if and only if it needs to, i.e. when it requests information on the object through a query call, as in `value := x.some_query`. This automatic mechanism is known as *wait-by-necessity* [4]. The lock-passing mechanism described in section 5 will slightly modify that policy: procedure calls that involve lock-passing will also require the client object to wait, as in the case of a query call.

### 3 Eliminating unnecessary locks

In this section, we take the first step towards relaxing the locking policy of SCOOP – we present a simple mechanism that allows the programmer to specify precisely which formal arguments of a routine should be locked. This allows us to eliminate the unnecessary locking – only the locks that are strictly necessary will be acquired. The mechanism relies on the concept of *detachable types* recently introduced in the Eiffel language [5]; it is fully compatible with other object-oriented concepts such as polymorphism, inheritance, and genericity.
3.1 (Too much) locking considered harmful

Recall that SCOOP requires that all separate arguments of a routine call be locked before the call can proceed. This policy is too restrictive and it unnecessarily increases the likelihood of deadlock. Consider the following feature \( r \).

\[
\begin{align*}
\text{local} & \quad \text{my}_y \text{. separate } Y \\
& \quad \text{my}_z \text{. separate } Z \\
\text{do} & \\
& \quad x \text{. } f \quad \text{-- separate call} \\
& \quad \text{my}_y \text{:= } y \\
& \quad x \text{. } g \quad \text{-- separate call} \\
& \quad \text{my}_z \text{:= } z \\
& \quad s \text{ } (z)
\end{align*}
\]

According to SCOOP, the processor(s) that handle \( x \), \( y \), and \( z \) must be locked by the client object before the body of \( r \) can be executed. Is it really necessary to lock all of them? Let’s see: the body of \( r \) contains two calls on \( x \), therefore \( x \) needs to be locked. There is no way around it – we must ensure that no other client is currently using \( x \). On the other hand, \( y \) only appears on the right-hand side of an assignment; no calls on \( y \) are made. Similarly, \( z \) only appears as source of an assignment and as actual argument of a feature call. It seems that we only need to lock the processor that handles \( x \); it is not necessary for \( y \) and \( z \) because the body of \( r \) does not contain any calls on them.

The eager locking applied by SCOOP might be very dangerous as it often leads to deadlocks – the more resources a client requires, the more likely it is to get in a deadlock situation. The example above shows that the locking policy can be easily refined to avoid these drawbacks.

3.2 Detachable types and their concurrent semantics

The attached type mechanism is an extension of Eiffel’s type system. Every type is declared either as “attached” or as “detachable”; an attached type guarantees that the corresponding values are never void. The default case is attached, e.g., \( x \): \( X \) means “\( x \) is of type attached \( X \)”. Detachable types are marked with ‘?’, e.g., \( y \): ? \( Y \) means “\( y \) is of type detachable \( Y \)”. A qualified call \( x \). \( f \) \( (a) \) is valid only if the type of \( x \) is attached. A new validity rule allows an attachment (assignment or argument passing) from the attached version of a type to the detachable version but not the other way round (unless a check of non-voidness is performed). We can rely on the use of detachable and attached types to specify which arguments of a routine should be locked. We require that all separate formal arguments that are locked by a routine be of an attached type. Conversely, all separate formal
arguments that need not be locked should be of a detachable type. Let’s apply the rule to the example used in the previous section. Now, only the processor that handles $x$ will be locked when a call to $r$ is executed. The processors that handle $y$ and $z$ will not be locked.

$$r \ (x: \text{separate } X; \ y: \ ?\text{separate } Y; \ z: \ ?\text{separate } Z)$$

```plaintext
local
  my\_y: \ ?\text{separate } Y
  my\_z: \ ?\text{separate } Z

do
  x. f
  my\_y \ := \ y
  x. g
  my\_z \ := \ z
  s \ (z)
end
```

Note that the applied rule is consistent with the general property of detachable and attached types: an entity needs to be attached only if we perform a call on it. Since no calls are made on $y$ and $z$, there is no need to declare them as attached (and therefore lock their processors).

### 3.3 Support for inheritance and polymorphism

Our technique is compatible with inheritance and polymorphism. Since $T$ is a subtype of $\ ?T$, we may redefine a feature in a descendant class following the rule for result and argument redefinition:

- The return type of a feature may be redefined (covariantly) from $\ ?T$ to $T$.
- The type of a formal argument may be redefined (contravariantly) from $T$ to $\ ?T$.

If the original version of the feature takes an argument of type $\text{separate } T$, we can redefine it in a descendant so that it takes an argument of type $\ ?\text{separate } T$. A client that uses the original class will need to pass an attached actual argument. Even if the redefined version of the feature is called (due to dynamic binding), that actual argument will satisfy the required type. Obviously, we cannot redefine a detachable formal argument into an attached one - the type safety would not be preserved in the presence of polymorphism and dynamic binding. Note that the contravariant redefinition rule for the “detachability” of formal arguments (as opposed to the covariant rule for their class types) implies that the redefined version of a feature may lock at most as many arguments as the original one. In other words, the clients will not be cheated on - they may expect at most as much locking as specified by the signature of the feature; no additional locking may be introduced when redefining the feature.
3.4 Discussion

In addition to the solution based on attached types, we considered two alternative ways of specifying which formal arguments should be locked. The first solution is a compiler optimisation: if the body of r does not perform any calls on x, then the processor handling x does not need to be locked. The programmer does not need to use any additional type annotations to mark the arguments to be locked. Unfortunately, this solution is not acceptable for two main reasons:

- The client cannot see whether the formal argument is locked or not without looking at the implementation of the feature; the interface is not precise enough to infer all the necessary information.
- In the presence of polymorphism and dynamic binding the client might be cheated on - a redefined version of the feature might lock an argument that the original version does not lock.

The second solution relies on the extensive use of pre-conditions. In order to make sure that the processor handling x is locked throughout the execution of r's body, we need to include the assertion \( is\_available \ (x) \) in the precondition clause:

```
r (x: separate X; y: separate Y; z: separate Z)
    require
        is_available (x)
    do ...
end
```

Such assertions are like wait-conditions (see 2.3) - they make the client wait until the processor that handles the corresponding formal argument is available (i.e. it can be locked). This solution is compatible with polymorphism and dynamic binding. Removing \( is\_available \ (x) \) from the precondition clause of a redefined version of r eliminates the lock requirement on x's processor. Such redefinition can be viewed as a particular case of precondition weakening which is a standard technique of Design by Contract. Although theoretically sound, this solution is not likely to be accepted in practice because it is too verbose and it puts too much burden on the programmer. Also, it is based on the special semantics for the assertion \( is\_available \) which might be a bit misleading - programmers might think that \( is\_available \) is a feature applicable to \textbf{Current}. Finally, as a matter of taste, it seems much easier to write (and read) code like this

```
s (x, y, z: separate X; a: ?separate A)
    do ...
end
```

using the technique based on attached types, than clumsy code like that

```
s (x, y, z: separate X; a: separate A)
```
The solution based on attached types is the only one that is theoretically sound, practical, and elegant. It also integrates best with other object-oriented mechanisms. We decided to adopt it as the standard approach.

4 Shared locking

After answering the question “which arguments should be locked at all?” that we addressed in the previous section, we take a more detailed look at the locking needs of clients and we try to answer the question: “when do we need an exclusive lock, when can we accept a shared lock?” We present a shared-locking mechanism that allows multiple clients to access simultaneously supplier object(s) handled by the same processor. We introduce the notion of pure query and use it as a basis for the new locking mechanism. The main motivation for the introduction of shared locking is to increase the expressiveness of the SCOOP model, i.e. to make it possible to write certain scenarios that were non-implementable in the original model. We also want to achieve some performance gain by avoiding unnecessary waiting. The mechanism is devised in such a way that multiple locking of separate objects preserves the integrity of the system and does not create the potential for additional deadlocks.

4.1 The need for sharing

In SCOOP, at most one client object may access the given supplier object at any time. If we consider the fact that processors are sequential, that is at most one feature is executed by a processor at any time, this ensures the absence of data races and gives the client a very strong guarantee of exclusive access to a processor. It simplifies the implementation and makes it easier to reason about concurrent programs. The biggest drawback of this approach is the loss of performance: very often, several clients try to access a shared data structure and only one of them is allowed to make a call at a time; the others have to wait, even if they do not try to modify the data structure. Consider a typical reader-writer scenario in Figure 1. Imagine that several client objects (some of them readers, some of them writers) call features read and write, respectively, on the same supplier object my_container. According to the SCOOP policy, all these calls will be executed in mutual exclusion. Certainly, this is necessary for writers’ calls because feature write modifies the state of the supplier object. On the other hand, the readers could be allowed to proceed in parallel because feature read does not modify the state of the supplier object. Unfortunately, this is not possible in SCOOP - multiple calls to read may be interleaved but
they cannot proceed in parallel. One might think that the loss of performance in that scenario is not that important, in particular on a single-CPU machine where there is no real parallelism and single instructions need to be interleaved anyway. It is true if feature `read` just retrieves an element from the container; it is not true anymore if some (local) computation is performed after retrieving the element:

```plaintext
read (a_container: separate CONTAINER [G]): G
  require
  not a_container.is_empty
  do
    Result := a_container.item
  end
end −− class READER [G]
```

Since the routine body represents a critical section w.r.t. the formal argument `a_container`, each reader will hold an exclusive access to the container for the whole duration of the routine call, also during the (potentially long-lasting) local computation. This will inevitably lead to a considerable loss of performance because all the other clients will be kept on hold. Again, one might argue that feature `read` might be split into two smaller features where only the first one would lock `a_container` to read an item form it, and the second one would just perform the (long-lasting) computation without locking any arguments. This is
true if we assume that there is no polymorphism and that the programmers have no possibility of redefining feature read in a subclass. Obviously, such assumptions go against the very basis of the object-oriented computation. As soon as we allow polymorphism and feature redefinition, we cannot make stronger assumptions about the given feature than what its contract says. Therefore, we might expect that even a “simple” feature like read might involve some local computations. Also, certain routines cannot be split into smaller chunks, e.g. if the result of the local computation is used as an argument of a second separate call on the locked formal argument. We need a simple way to specify which formal arguments a routine should be locked for the exclusive use by the client and which ones can be locked in a shared mode where multiple clients are allowed to access the given processor at the same time. While deciding whether an argument should or should not be locked at all amounts to a simple type check (see section 3), deciding about the kind of lock that is required is more complex. We must make sure that multiple client accesses to a processor do not violate the consistency of the object structure handled by that processor. Also, the guarantees given to a client should be preserved, independently of the number of other clients that happen to be using the same processor in the shared mode. In fact, a client should not be aware of the presence of other clients.

4.2 Interleaving

We looked for a suitable mechanism to support simultaneous access to supplier objects. After the unsuccessful attempts at using the intra-object concurrency for that purpose (see section 4.6), we decided to preserve the sequential character of SCOOP processors but allow interleaving of features requested by different client objects. We realised that certain feature calls may be interleaved without violating the guarantees given to clients; in fact, we can demonstrate that such interleavings are serialisable [6], i.e. the effect of executing them is the same as if the involved features are executed one after another. Consider the example in Figure 2. Assume that my_list denotes the same object for both clients. Client 1 wants to evaluate the sum of elements number 5 and 6 from my_list, then remove the fourth element from the list. At the same time, client 2 wants to evaluate the sum of elements number 4 and 5 from the same list. If we apply the standard locking policy of SCOOP, i.e. required that each client acquire an exclusive lock on the processor that handles my_list, that processor might execute the following sequence of features:

\begin{verbatim}
−− code executed for client 1−− code executed for client 2
(1) i_th (5)          (2) i_th (5)
(2) i_th (6)          (3) i_th (4)
(3)                   (4) i_th (5)
(4) cursor           (5) cursor
(5) go_to_i_th (4)    (6) go_to_i_th (4)
(6) prune            (7) prune
\end{verbatim}
As you can see, the feature calls corresponding to the invocations of \textit{sum of two} (i.e. two subsequent applications of \textit{i\_th}) and \textit{remove\_i\_th} are not interleaved; they are executed in a strict sequence (\textit{sum\_of\_two} on behalf of client 1, then \textit{sum\_of\_two} on behalf of client 2, then \textit{remove\_i\_th} on behalf on client 1). We relax that policy to allow the interleaving of the feature bodies provided that such interleaving can be serialised. Figure 3 lists three examples of interleaving. Figure 2. Two clients using a shared list

As you can see, the feature calls corresponding to the invocations of \textit{sum of two} (i.e. two subsequent applications of \textit{i\_th}) and \textit{remove\_i\_th} are not interleaved; they are executed in a strict sequence (\textit{sum\_of\_two} on behalf of client 1, then \textit{sum\_of\_two} on behalf of client 2, then \textit{remove\_i\_th} on behalf on client 1). We relax that policy to allow the interleaving of the feature bodies provided that such interleaving can be serialised. Figure 3 lists three examples of interleaving.

In the interleavings a) and b) the execution of instructions (applications of \textit{i\_th}) that correspond to the calls to \textit{sum\_of\_two} issued by different clients is overlapped. \textit{i\_th} is a feature without side effects so the result of these interleavings is the same as if the calls are executed in a sequence, without overlapping. We say that these interleavings are serialisable. On the other hand, interleaving c) is not serialisable because there exists no sequential execution that would

\begin{verbatim}
-- in class CLIENT
sum_of_two ( a_list: separate LIST [INTEGER]; i: INTEGER): INTEGER
  require
    a_list . count > i
  local
    e1, e2: INTEGER
  do
    e1 := a_list . i\_th (i)
    e2 := a_list . i\_th (i + 1)
    Result := e1 + e2
  end

remove\_i\_th ( a_list: separate LIST [INTEGER]; i: INTEGER)
  require
    a_list . count >= i
  local
    pos: CURSOR
  do
    pos := a_list . cursor
    a_list . go\_i\_th (i)
    a_list . prune
    a_list . go\_to (pos)
  end

-- client 1
sum := sum_of_two (my_list, 5)
remove\_i\_th (my_list, 4)

-- client 2
sum := sum_of_two (my_list, 4)
\end{verbatim}
yield the same result. This is caused by the overlapping of `remove_i.th` (a feature with side effects) and `sum_of_two`. It would seem that we found the solution: we should allow serialisable interleavings and disallow non-serialisable ones. Unfortunately, the problem of serialisability is undecidable in the presence of aliasing and polymorphism and without the closed-world assumption. Nevertheless, our analysis gives us a very useful hint: it is easy to see that any interleaving of side-effect-free features is serialisable; it is not the case (in general) for feature calls with side effects.

### 4.3 Pure queries and shared locks

Following that observation, we need to restrict the set of valid interleavings to those where only side-effect-free features are interleaved; other feature calls must
be executed in a strict sequence. We require that features with side effects be only applied to a supplier object whose handling processor has been locked by the client object in an exclusive mode. This eliminates the danger of invalid interleaving. Side-effect-free features can be applied to a client object whose handling processor has been locked by the client object in a shared mode or an exclusive mode.

To decide what locking mode is necessary for a given feature call, we would like to rely on the standard notion of side-effect-free feature (or pure function) as used in the sequential context. Pure functions are only allowed to temporarily modify the state of the supplier - they must make sure that the state of the supplier after the execution of the query is the same as its state before the execution. Obviously, attributes are pure; so are functions that only modify local variables. Creation of new objects is not considered as a side-effect so it does not influence the “purity” of queries. Feature $i_{th}$ is an example of a pure function - it preserves the state of the supplier object even though it modifies it temporarily using features $go_{i_{th}}$ and $go_to$.

In Eiffel, in order to declare a feature as a pure function, it is sufficient to include an empty only clause in its post-condition. The only clause always appears as the last postcondition clause of a feature. It lists all the entities that the given feature is allowed to modify. An empty only clause states that the feature does not modify any entities. Unfortunately, this standard notion of purity is too imprecise to be applied in the concurrent context. The main problem is that it only allows to make assumptions about the state of the supplier object but nothing can be said about the state of other objects that are referenced by the supplier. For example, feature $remove_{i_{th}}$ introduced in Figure 2 can be considered as pure:

```eiffel
remove_{i_{th}} ( a_list : separate LIST [INTEGER]; i: INTEGER)
  require
    a_list . count >= i
  local
    pos: CURSOR
  do
    pos := a_list . cursor
    a_list . go_{i_{th}} (i)
    a_list . prune
    a_list . go_to (pos)
  ensure
    only /* empty only clause */
end
```

The state of $Current$ is not modified. Only the state of local variable $pos$ and the state of the object referred to by $a_list$ change. Note that even the latter does not violate the standard purity assumption: the reference $a_list$ still has the same value (it points to the same object); only the object that is referenced has been modified. The purity of $remove_{i_{th}}$ implied by its empty only clause may suggest that the feature has no side effect on $a_list$ and therefore a shared lock
on \texttt{a_list}'s processor would be sufficient. Clearly, this is not the case - the state of the object represented by \texttt{a_list} is modified so an exclusive lock is necessary.

We refine the notion of purity so that it takes into account the target w.r.t. which a feature is side-effect-free. The solution relies on the enhanced type system for SCOOP introduced in [7] and further developed in [8]. Let us use the following example to introduce the new notation for specifying the relative purity of features:

\begin{verbatim}
copy_and_append (a_source: separate LIST [G]; a_target: separate LIST [G]; i: INTEGER)
   -- Read element number \(i\) from \texttt{a_source}
   -- and append it to \texttt{a_target}.
   require
      a_source.count \(\geq\) i
   do
      a_target.append (a_source.i.th (i))
   ensure
      only <a_target.handler>
\end{verbatim}

Feature \texttt{copy_and_append} may only modify the state of the object structure handled by \texttt{a_target}'s processor, as expressed by the \texttt{only} clause (note the use of \texttt{<a_target.handler>} to denote “the processor that handles \texttt{a_target}”). It does not modify the state of the object structure handled by \texttt{a_source}'s processor (we assume that feature \texttt{i.th} in class \texttt{LIST [G]} has an empty \texttt{only} clause); neither does it modify the state of the object structure handled by \texttt{Current}'s processor. We can say that \texttt{copy_and_append} is pure w.r.t. \texttt{<Current.handler>} and \texttt{<a_source.handler>}. Therefore, it will only lock \texttt{a_target}'s processor in the exclusive mode; \texttt{a_source}'s processor will be locked in shared mode. It is very important to reason about the whole object structure that is handled by a given processor because locking happens at the level of processors rather than single objects. For example, we know that the call to \texttt{a_source.i.th} will not modify the state of \texttt{a_source} but if it is allowed to modify other objects that are handled by \texttt{a_source}'s processor, we cannot treat \texttt{copy_and_append} as pure w.r.t. \texttt{<a_source.handler>} anymore. We are now ready to define the notion of \textit{pure query} and give the refined rules for locking.

\textbf{Definition 1. Pure query} Feature \(f\) is a \textit{pure query} w.r.t. processor \(\alpha\) iff it satisfies the following conditions:

\begin{enumerate}
   \item \(\alpha\) has an explicit \texttt{only} clause \texttt{(onlyf)}
   \item \(<\alpha>\not\in\texttt{onlyf}\)
   \item \(\neg\exists e :: (\alpha,T) \land \texttt{<e.handler>} \in \texttt{onlyf}\)
   \item every feature \(g\) called by \(f\) using an unqualified call is a pure query w.r.t. \(\alpha\)
   \item every feature \(h\) called by \(f\) using a qualified call on target \(x\), where \(x :: (\alpha,T)\), is a pure query w.r.t. \texttt{<Current.handler>}, where \texttt{<Current.handler>} is evaluated in the context of \(h\)
\end{enumerate}
Note that $\alpha$ may be an explicit processor tag or a processor expression, i.e. an entity name followed by “.handler”, e.g. <$a_{source}\_handler$>. Only constant entities (such as formal arguments and Current) may be used as target of processor expressions. Pure queries may create new objects on the processor as long as this operation does not modify the existing object structure handled by the processor. The correctness property for pure queries is stronger than the corresponding property for other features:

Property 1. If $f$ is a pure query w.r.t. processor $\alpha$, and the body of $f$ is executed when the pre-condition of $f$ and the invariant of the underlying class hold, then:

- the post-condition of $f$ and the invariant are satisfied after the call.
- the state of the object structure handled by processor $\alpha$ does not change (modulo object creation) during the execution of $f$ at any observable state.

Property 1a is the standard correctness requirement for features in the sequential context. Property 1b allows for safe interleaving of pure query calls. Since the state of the object structure handled by the processor $\alpha$ does not change (it may change temporarily but such changes are not observable to the clients), any interleaving of clients’ calls can be serialised. Therefore, clients only need to lock that processor in a shared mode.

Rule 2. Locking.

- If feature $f$ is a pure query w.r.t. processor $\alpha$, and call to $f$ needs to lock processor $\alpha$, then $f$ requests a shared lock on that processor.
- If feature $f$ is not a pure query w.r.t. processor $\alpha$, and call to $f$ needs to lock processor $\alpha$, then $f$ requests an exclusive lock on that processor.
- A shared lock on a processor can be acquired even if another client object is currently holding a shared lock on the same processor. If some client holds an exclusive lock on a processor, other clients can obtain neither a shared lock nor an exclusive lock on that processor.

Example. Suppose that several client objects try to access the same shared buffer (implemented as a bounded queue). Three operations are allowed: storing a value into the buffer, reading a value from the buffer, and consuming a value from the buffer. These operations are implemented as features store, read, and consume (see Appendix). Feature read does not change the state of the buffer at any point of its execution, it only returns the oldest element stored. Therefore, read is a pure query w.r.t <$a\_buffer\_handler$>. Although consume is a query implemented as a function, it is not a pure query because it changes the state of the object referred to by $a\_buffer$. Also store modifies the state of $a\_buffer$. According to the refined locking rules, an exclusive lock must be obtained for calls to store and consume; a shared lock is sufficient for calls to read. Suppose that we execute the following sequence of calls (where $o1$, $o2$, $o3$, $o4$, $o5$, $o6$, and $my\_buffer$ are handled by different processors):
When the execution of the sequence above begins, $o1$ is granted a shared lock on $my\_buffer$ and it can execute feature $read$. Since $o5$ is requesting an exclusive lock to execute $store$, it must wait until $o1$ has released its lock. Now $o2$ is requesting a shared lock on $my\_buffer$, so it can acquire the lock and execute $read$. Since $o1$ and $o2$ only hold a shared lock on $my\_buffer$, $o3$ does not have to wait: it is granted a shared lock and can proceed with the execution of $read$. Similarly, $o4$ proceeds without waiting. The four calls to $read$ are executed in parallel, i.e. the applications of feature $item$ requested by the clients may be interleaved in any order. The client object $o6$ is now requesting an exclusive lock in order to execute $consume$; it must wait until all shared locks have been released and all previous requests for exclusive locks have been satisfied. In this case, $o5$’s request will be satisfied first, followed by $o6$’s request.

With the single locking policy currently used in SCOOP, all the calls would have been executed in a strict sequence. Even the calls to the pure query $read$ would have been serialised. This simple example shows that the refined locking policy can bring a considerable performance gain to SCOOP-based programs. This is especially important in applications where many concurrent accesses to shared data structures are executed, e.g. in intensive scientific computations. Nevertheless, one has to be very careful when designing the scheduling policy so that the scheduling is fair. In the example above, clients requesting shared locks were “privileged” by the scheduler; clients requesting exclusive locks had to wait a bit longer. This problem is discussed more thoroughly in section 4.5.

### 4.4 Locking requirements at run-time

What happens when a feature has different locking requirements for different formal arguments but the corresponding actual arguments happen to be handled by the same processor? Consider the situation where $copy\_and\_append$ is called with the actual arguments corresponding to $a\_source$ and $a\_target$ being handled by the same processor:

```plaintext
my\_source\_list: separate <p1> LIST [INTEGER]
my\_target\_list : separate <p1> LIST [INTEGER]
...

copy\_and\_append (my\_source\_list, my\_target\_list, 1)
```

We know that $my\_source\_list$ and $my\_target\_list$ are handled by the same processor – an explicit processor tag $p1$ is used in type declarations of both entities (see [8] for a more in-depth discussion on processor tags). Since feature $copy\_and\_append$ is pure with respect to the processor that handles its first argument, it will only request a shared lock on $p1$. Nevertheless, it will need an exclusive lock on the processor that handles its second argument, which happens to be $p1$ again. The locking mechanism will always apply the weakest lock that satisfies all the requirements of the feature w.r.t. the given processor. In this
particular case, an exclusive lock will be acquired on \( pI \) – a shared lock would be too weak.

Note that current processor (\(<\texttt{Current.handler}>\)) is always considered to be locked in the exclusive mode, even if the feature does not require it explicitly.

### 4.5 Related Issues

Extending the access control policy comes at a price. Several aspects of the SCOOP model are influenced by the new locking scheme:

**Deadlock Prevention.** The deadlock prevention scheme becomes more complex. In particular, potential deadlocks resulting from the “lock upgrade” requests (from shared to exclusive) might be very difficult to analyse and eliminate at compile-time.

**Scheduling Policy.** The current scheduling policy in SCOOP is very fair: a separate call is scheduled as soon as all the necessary locks can be acquired and the wait-conditions are satisfied. Locks are granted in a FIFO order; there is no starvation. Implementation of such policy is straightforward – see SCOOPLI library [9]. Implementation of a similar scheduling mechanism with the refined locking policy is much more difficult. We have to deal with the *writer starvation* problem. An implementation may decide to check if a lock request is compatible with the current locking state of the processor and immediately grant the requested lock if the request is compatible (i.e. a client requests a shared lock on a processor that is already locked in a shared mode). Such policy may cause writer starvation problem if there are a lot of shared lock requests. If a client object was granted a shared lock, then other clients requesting a shared lock can immediately join in. However, clients requesting an exclusive lock (writers) will be blocked out, until all readers have finished. In fact, some unlucky clients may get blocked infinitely.

**Compiler Support.** Currently, the ISE Eiffel compiler\(^1\) does not support the concept of pure queries. This means that the purity of a query cannot be automatically checked at compile time. The compiler has to be extended so that, if a feature is declared as pure w.r.t. a certain processor (i.e. the “syntactic” conditions a), b), and c) in Definition 1 are satisfied), the feature is indeed pure (i.e. the “semantic” conditions d) and e) hold).

### 4.6 Discussion

In our search for a suitable mechanism for supporting shared locking, we investigated the possibility of allowing intra-object concurrency in SCOOP. First

\(^1\) Available from [http://www.eiffel.com](http://www.eiffel.com)
of all, it implies that processors be multi-threaded. To prevent interference, we
would only allow non-conflicting features to be executed in parallel on the same
supplier object. Unfortunately, the problem of non-interference of features is not
decidable in the presence of aliasing and polymorphism. Therefore, we initially
proposed a very restricted solution that would only allow so-called pure func-
tions to be executed in parallel. Nevertheless, even that solution was not sound
if we considered the standard notion of purity applied in object-oriented lan-
guages (no side-effects except for temporary ones). The main problem was the
impossibility of ensuring that the invariant of the class and the post-condition of
a feature hold if another pure feature executed in parallel temporarily modifies
the state of the supplier object. Consider feature \texttt{i-it} from class \texttt{LIST}\,[G]:

\begin{verbatim}
\begin{verbatim}
i\_th (i: INTEGER): G
    -- Item at 'i'-th position
    local
        pos: CURSOR
    do
        pos := cursor
        go_i_th (i)
        Result := item
        go_to (pos)
end
\end{verbatim}
\end{verbatim}

\texttt{i-th} stores the current cursor position in a local variable \texttt{pos}, advances the cursor
to position \texttt{i}, performs a read operation (\texttt{item}), and moves the cursor back to
its initial position. Since the state of the supplier object does not change, this
feature should be considered as pure (in section 4.2 we saw that interleaving of
several calls to \texttt{i-th} does not lead to atomicity violations). Unfortunately, in the
presence of intra-object concurrency, its execution may lead to a post-condition
violation (or an invariant violation) if another feature (even a pure query, e.g.
\texttt{item}) has been executed and terminated before the cursor is moved back to its
initial position. Consider the following situation where client 1 is calling feature
\texttt{i-th} and client 2 is calling feature \texttt{item} on the same supplier object:

\begin{verbatim}
(1)  pos := cursor       item
(2)  go_i_th (i)        -- evaluate post-condition
(3)  Result := item     -- cursor = old cursor
(4)  go_to (pos)
\end{verbatim}

If the call to \texttt{item} on behalf of client 2 is executed before the call to \texttt{go_i_th}
on behalf of client 1, and the post-condition of \texttt{item} is evaluated before the call
to \texttt{go_to}, then that post-condition (\texttt{cursor = old cursor}) will be violated - the
value of \texttt{cursor} recorded on entry to the feature call (denoted by \texttt{old cursor}) will
be different from the value of \texttt{cursor} recorded at the exit (which is still equal to
\texttt{i} as a result of \texttt{go_i_th (i)} executed on behalf of client 1). Note that the post-
condition of item is violated even though both executed features are pure functions! We initially believed that enforcing a stronger notion of purity would solve the problem. For example, we would only consider attributes as pure enough to be evaluated in parallel. Nevertheless, in the presence of polymorphism, if the redefinition of an attribute into a routine is allowed, even attributes do not provide a sufficient guarantee of non-interference. The solution that uses intra-object concurrency suffers from additional drawbacks. First, reasoning about programs that allow intra-object concurrency is inherently more complicated. Secondly, providing a stronger notion of purity in the concurrent context would contradict the overall approach to object-oriented concurrency that we took - we want to find a more general definition of object-oriented constructs that would reduce to the usual sequential semantics, and not the other way round. Finally, sequential code would not be reusable in a concurrent context, so programmers would need to write their code so that it is “concurrency-friendly”. This would contradict the SCOOP approach of complete fusion of concurrency and object-orientation. Also, most of the existing code would be useless in a concurrent context (e.g. most data structures in the EiffelBase library).

5 Lock passing

Another refinement of the access control policy that increases the expressivity of the model is the lock passing mechanism. It allows clients to pass on their locks to their suppliers when needed. Such a scenario was impossible to implement in the original SCOOP model. The mechanism also makes concurrent programs less deadlock-prone.

5.1 The need for lock passing

Consider the following example:

\[
\begin{array}{l}
\text{r (x: separate X; y: separate Y)} \\
\text{do} \\
\quad x.f \\
\quad x.g (y) \quad \text{-- x waits for y to become available.} \\
\quad y.f \\
\quad \ldots \\
\quad z := x.some_query \quad \text{-- Current waits for x.} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{-- DEADLOCK!} \\
\text{end}
\end{array}
\]

Calls to \( x.f \), \( x.g \), and \( y.f \) are asynchronous (\( f \) and \( g \) are commands), so the client will not wait for their completion. In fact, following the \textit{wait-by-necessity} principle (see section 2.3), the client will only wait for the result of the query call \( x.some_query \). Unfortunately, this will cause a deadlock because the processor that handles \( x \) will not be able to evaluate \( some_query \) before finishing all the previously requested calls on \( x \); it will not be able to execute \( x.g (y) \) until it
acquires a lock on the processor handling $y$ but that processor is still locked by
the client and it can only be unlocked once the client finished the execution of
$r$’s body. So, the client is waiting for $x$’s processor and vice-versa; none of them
will ever make any progress.

In fact, getting into a deadlock situation is even simpler. The client may
simply pass itself as an actual argument to a separate query call:

$$
s (x: \text{separate } X)
\quad \begin{aligned}
& \text{do} \\
& \quad z := x.g (\text{Current}) \quad -- x \text{ waits for Current; Current waits for } x. \\
& \quad -- \text{ DEADLOCK!}
\end{aligned}
\end{equation}
$$

In both examples, we could avoid the deadlock if the client was able to tem-
porarily pass on the lock on $<y.\text{handler}>$ (respectively $<\text{Current}.\text{handler}>$) to
$x$.

### 5.2 The mechanism

We introduce the lock passing rule based on a new semantics for argument pass-
ing and attached types.

**Rule 2. Lock passing.** Assume that $x$, $y$, and $c$ are handled by different
processors ($P_1$, $P_2$, and $P_3$, respectively). If client $c$ holds a lock on $P_1$ and $P_2$,
and $c$ makes a separate call $x.f (y)$ then, if $y$ is of attached type, the call will
be executed synchronously, with $c$ passing on all its locks to $x$ and waiting until
the execution of $f$ terminates, then revoking all its locks from $x$ and continuing
its own execution. Let us revisit the examples.

$$
r (x: \text{separate } X; y: \text{separate } Y)
\quad \begin{aligned}
& \text{do} \\
& \quad x.f \\
& \quad x.g (y) \quad -- \text{ Current passes its locks to } x \\
& \quad -- \text{ and waits until } g \text{ terminates.} \\
& \quad y.f \\
& \quad \ldots \\
& \quad z := x.\text{some_query} \quad -- \text{ No deadlock here!}
\end{aligned}
\end{equation}
$$

We are able to avoid the deadlock situation in routine $r$ – the call to $x.g (y)$ is
executed synchronously and $x$ does not need to wait for a lock on $y$ since it gets it
through lock passing.

Similarly, routine $s$ is not deadlock-prone anymore because $x.g (\text{Current})$
results in a lock passing that allows $x$ to get a lock on $<\text{Current}.\text{handler}>$
without waiting:

$$
s (x: \text{separate } X)
\quad \begin{aligned}
& \text{do}
\end{aligned}
\end{equation}
$$
\[ z := x \cdot g \textbf{(Current)} \quad \text{-- x gets lock on Current from Current.} \]

\[ \text{-- No deadlock here!} \]

end

Note that, whenever a lock-passing occurs as a result of a feature call, the client passes \textit{all} its locks to the supplier, not only the locks on processors that handle the objects corresponding to the actual arguments of the call. This is because it does not matter to the client how many locks have been passed since it is blocked anyway until the execution of the feature has terminated; on the other hand, the supplier might require these additional locks in order to terminate the execution of the feature. Therefore, all locks are passed “just in case”. Such generous behaviour of clients often allows to avoid potential deadlocks.

6 Related work

Meyer [10] discusses detachable types, in particular their use for eliminating \textit{catcalls}. He also describes the idea of using detachable types in the context of SCOOP that was a result of our earlier discussions. He does not discuss the problem of feature redefinition in a concurrent context. Nevertheless, his solution of the catcall problem reveals a problem that is also relevant to SCOOP. To prevent catcalls, Meyer’s rule for argument redefinition requires that if the class type of a formal argument is redefined covariantly, it must become detachable. No restrictions are put on formal arguments that appear as call targets in inherited preconditions and postconditions; inherited assertions involving calls on detachable targets are evaluated using an implicit object test, e.g. \( x\.is_empty \quad \text{and} \quad y\.is_empty \) becomes \( x\.is_empty \textbf{ and } y\.is_empty \textbf{ and } \{y': Y\} y \textbf{ implies } y'.is_empty \), hence \( x\.is_empty \textbf{ if } y \textbf{ is void and } x\.is_empty \textbf{ and } y\.is_empty \textbf{ otherwise}. This is problematic because it may lead to postcondition weakening. We are currently working on a solution that is compatible with our locking policy but prohibits postcondition weakening.

Brooke et al. [2] propose a CSP semantics for SCOOP. In their model, locking is transitive by default, i.e. if client object \( c \) holds locks on supplier objects \( x \) and \( y \) and \( x \) requests a lock on \( y \), \( x \) will acquire that lock. Transitive locking has been proposed as a way to clarify the semantics of locking in SCOOP rather than an extension of the model. The main differences w.r.t. our mechanism are as follows:

- No synchronisation between calls on \( y \) issued by \( c \) and \( x \) takes place, i.e. it is possible that these calls are interleaved. This offers more potential parallelism than our solution (we apply synchronous semantics to calls that involve lock passing) but reasoning about programs becomes much more complex because one has to account for possible interleavings.
- The client cannot decide whether to pass locks or not; transitive locking is always applied. In our approach, the programmer decides whether lock passing should take place; it can be achieved through different combinations of detachable and attached types of feature arguments.
– It is possible for client $c$ to unlock supplier $y$ before supplier $x$ revokes its lock on $y$. Our mechanism disallows such behaviour; locks that have been passed to a supplier must be revoked before the client may proceed.

– Only locks on separate objects are passed to the client; passing Current (or another non-separate entity) as actual argument of a separate call does not result in supplier processor getting a lock on the client’s processor.

– Only one lock is passed on to the supplier. In our approach, all locks held by the client are passed to minimise the risk of potential deadlock in case the supplier requires other locks later on.

We would be interested in a CSP formalisation of our approach. The formal model proposed by Brooke et al. may be extended to account for the differences mentioned above. In particular, every call involving lock passing should be treated similarly to a query call, i.e. it should be executed synchronously, with the client’s handler being blocked until the call returns.

7 Conclusions

We presented three simple refinements of the access control policy for SCOOP. First, we proposed a mechanism for specifying which arguments of a routine call should be locked. This mechanism is based on the novel concept of attached types. Secondly, we introduced the notion of pure query and devised an optimistic locking policy based on it. The locking policy allows for shared locking of processors and interleaving of pure query calls issued by different clients. Finally, we introduced a lock-passing mechanism that reduces the danger of deadlocks. The proposed mechanisms greatly improve the flexibility of the model. They allow programmers to efficiently implement synchronisation scenarios that were difficult (or even impossible) to implement in the original SCOOP model.

We have already implemented the lock-passing scheme. A deadlock-detection mechanism that supports lock-passing has also been devised and implemented [11] as part of the SCOOPLI library. We expect the full implementation of the refined access control policy to be completed within six to eight months. Our future research will be focused on the enhanced type system for SCOOP and its applications to:

– Reasoning about locality of objects.
– Integration of expanded types and agents with SCOOP.
– Anti-deadlock policy.

References

5. ECMA: Eiffel analysis, design, and programming language. ECMA Standard 367 (2005)
APPENDIX

Implementation of the producer-consumer-reader example from section 4.3

store (a_buffer: separate BUFFER [G]; value: G)
   -- Store value into a_buffer.
   require
      a_buffer_specified : a_buffer /= Void
      a_buffer_not_full : not a_buffer.is_full
      value_specified : value /= Void
   do
      a_buffer.put (value)
   ensure
      a_buffer_not_empty: not a_buffer.is_empty
      only <a_buffer.handler>
end

consume (a_buffer: separate BUFFER [G]): G
   -- Consume value from a_buffer.
   require
      a_buffer_specified : a_buffer /= Void
      a_buffer_not_empty: not a_buffer.is_empty
   do
      Result := a_buffer.item
      a_buffer.remove
   ensure
      correct_value: Result = old a_buffer.item
      one_element_removed:
         a_buffer.count = old a_buffer.count - 1
      a_buffer_not_full : not a_buffer.is_full
      only <a_buffer.handler>
end

read (a_buffer: separate BUFFER [G]): G
   -- Read value from a_buffer.
   require
      a_buffer_specified : a_buffer /= Void
      a_buffer_not_empty: not a_buffer.is_empty
   do
      Result := a_buffer.item
   ensure
      correct_value: Result = a_buffer.item
      only
end