

# Flexible access control policy for SCOOP

Piotr Nienaltowski

Praxis High Integrity Systems Limited  
20 Manvers Street  
Bath BA1 1PX  
United Kingdom

**Abstract.** The SCOOP model extends Eiffel to support the construction of concurrent applications with little more effort than sequential ones. The model provides strong safety guarantees: mutual exclusion and atomicity at the routine level, and FIFO scheduling of clients' calls. Unfortunately, in the original proposal of the model (SCOOP\_97) these guarantees come at a high price: they entail locking all the arguments of a feature call, even if the corresponding objects are never used by the feature. In most cases, the amount of locking is higher than necessary. Additionally, a client that holds a lock on a given processor cannot relinquish it temporarily when the lock is needed by one of its suppliers. This increases the likelihood of deadlocks; additionally, some interesting synchronisation scenarios, e.g. separate callbacks, cannot be implemented. We propose two refinements of the access control policy for SCOOP: a type-based mechanism to specify which arguments of a routine call should be locked, and a lock passing mechanism for safe handling of complex synchronisation scenarios with mutual locking of several separate objects. When combined, these refinements increase the expressive power of the model, give programmers more control over the computation, and enable more potential parallelism, thus reducing the risk of deadlock.

**Keywords:** concurrency; object-oriented programming; Design by Contract; SCOOP; attached types; locking; callbacks

## 1. Introduction

The SCOOP model, initially proposed by Meyer [Mey97] and subsequently refined and implemented in [Nie07], 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 (DbC) which have proved highly successful in improving the quality of sequential programs [Mey92], and extend them in a minimal way to cover concurrency and distribution. Concurrency in SCOOP relies on the basic mechanism of object-oriented computation: the feature call. Each object is handled by a *processor* — a conceptual thread of control — referred to as the object's *handler*. All features of a given

object are executed by its handler, i.e. only one processor is allowed to access the object. Several objects may have the same handler; the mapping between an object and its handler does not change over time. If the client and the supplier objects are handled by the same processor, the feature call is synchronous; if they have different handlers, the call becomes asynchronous, i.e. the computation on the client's handler may move ahead without waiting. Objects handled by different processors are called *separate*; objects handled by the same processor are *non-separate*. A processor, together with the object structure it handles, forms a sequential system. Therefore, every concurrent system may be seen as a collection of interacting sequential systems; conversely, a sequential system may be seen as a particular case of a concurrent system (with only one processor).

Since each object may be manipulated only by its handler, there is no object sharing between different threads of execution (no shared memory). Given the sequential nature of processors, this results in the absence of intra-object concurrency: there is never more than one action performed on a given object at a given time. Therefore, programs are data-race-free by construction. Locking is used to eliminate *atomicity violations*, i.e. illegal interleaving of calls from different clients. For a feature call to be valid, it must appear in a context where the client's processor holds a lock on the supplier's processor. Locking is achieved through the refined mechanism of *feature application*: the processor executing a routine blocks until the processors handling the objects represented by the actual arguments have been locked (atomically) for its exclusive use; the routine serves as a critical section. Since a processor may be locked by at most one other processor at any time, and all feature calls to objects handled by the same processor are executed in a FIFO order, no harmful interleaving occurs.

The lock-based access control policy provides strong safety guarantees but the price to pay is relatively high: all arguments of a feature call have to be locked, even if they are never used by the feature as targets of calls. Such unnecessary locking limits parallelism and increases the likelihood of deadlocks. Furthermore, a client holding a lock cannot relinquish it temporarily when the lock is needed by one of its suppliers. As a result, certain interesting concurrency scenarios require convoluted implementation patterns; other scenarios, e.g. separate callbacks, cannot be implemented at all. We propose two ways of relaxing the strict locking policy to solve the above problems: (1) a type-based mechanism to specify which arguments of a routine call should be locked, and (2) a lock passing mechanism, related to that introduced in [BPJ07], for safe handling of mutual locking between several separate objects.

The rest of the article is organised as follows. Section 2 analyses the access control policy of SCOOP and introduces the mechanism for selective locking. (The problems of precondition weakening and precursor calls discussed there are not concurrency-specific; their analysis and the proposed solutions may be regarded as contributions to DbC in general.) Section 3 introduces the lock-passing mechanism; Section 4 discusses its importance for proofs of software correctness. Section 5 presents related work. Finally, Section 6 concludes and points out future research directions.

In the rest of this article, we will refer to the original model proposed in [Mey97] as *SCOOP-97*, and to the refined model presented in [Nie07] simply as *SCOOP*.

## 2. Eliminating unnecessary locks

The access control policy of SCOOP-97 requires all formal arguments of a feature to be reserved before the feature is applied, as expressed by the Feature Application Rule (Definition 1). The processor applying the feature to a target object blocks until all the processors handling the actual arguments have been reserved.

**Definition 1 (Feature application rule).** Before a feature is applied, its formal arguments must be reserved by the supplier's handler, and its precondition must hold.

But this rule is too restrictive, as illustrated in Figure 1. The handlers of  $x$ ,  $y$ , and  $z$  must be locked on behalf of the executing processor before the body of  $r$  is executed (`some_precondition` must also hold before executing  $r$  but for the moment we will ignore this requirement; we will come back to it in section 2.2). Is it really necessary to lock all the arguments? 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 processor is currently using  $x$ . On the other hand,  $y$  only appears as source 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 only the processor that handles  $x$  needs to be locked; locking  $y$  and  $z$  is not necessary because the body of  $r$  does not perform any calls on them.

---

```

r (x: separate X; y: separate Y; z: separate Z)
  require
    some_precondition
  local
    my_y: separate Y
    my_z: separate Z
  do
    x.f
    my_y := y
    x.g
    my_z := z
    s (z)
  end

```

---

**Fig. 1.** Greedy locking

The greedy locking policy has several drawbacks. Most importantly, it increases the danger of deadlocks: the more resources a routine requires, the more likely it is to end up in a deadlock. It also reduces opportunities for parallelism and makes it impossible to pass references around without locking the corresponding objects; as a result, programmers have insufficient control over the locks.

## 2.1. Relaxed locking policy based on attached types

Our solution to the above problems relies on the use of *attached types* recently introduced in Eiffel<sup>1</sup> [ECM05, ISO06]. 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 accept `Void` as legal value. 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. The type rules of SCOOP cater for attached types and allow 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 dynamic check of non-voidness is performed); see [Nie07, Chapter 6]. Attached types can be used to specify which arguments of a routine should be locked. We only require that processors handling the objects represented by attached formal arguments of a routine be locked; those handling detachable formal arguments are not locked. This new semantics of attached and detachable types is not a mere “hack” to optimise locking. It follows the intuitive understanding of call validity in SCOOP: a client is allowed to perform a feature call if and only if the target is non-void and the client has exclusive access to the target’s processor, as expressed by the Call Validity Rule [Nie07, Definition 6.5.3].

The refined Feature Application Rule below captures precisely the necessary and sufficient conditions for a safe application of features.

**Definition 2 (Feature application rule (refined)).** Before a feature is applied, its attached formal arguments must be reserved by the supplier, and its precondition must hold.

This rule, combined with the Call Validity Rule, ensures atomicity: a routine `r` represents a critical section with respect to all the processors that handle its attached formal arguments. All the calls on targets handled by those processors within the body of `r` are guaranteed to be handled atomically; calls nested in the bodies of the routines called in `r`, however, are not guaranteed to be atomic, unless their target is controlled in `r`; see Definition 6. Therefore, the notion of atomicity used here is slightly weaker than the common understanding of the term. (Atomicity is usually understood as the complete absence of interference from other concurrently executing processors.)

Let us rewrite the example from Figure 1 to make use of the new mechanism. Figure 2 shows an optimised version of the routine `r`. Following Definition 2, the handler of `x` is locked before `r` is executed because `x` is an attached formal argument; the handlers of `y` and `z` are not locked because `y` and `z` are detachable.

---

<sup>1</sup> *Spec#* has a similar type mechanism: *non-nullable types* [FL03].

---

```

r (x: separate X; y: ?separate Y; z: ?separate Z)
  local
    my_y: ?separate Y
    my_z: ?separate Z
  do
    x.f
    my_y := y
    x.g
    my_z := z
    s (z)
  end

```

---

**Fig. 2.** Selective locking

The new rule for feature application gives programmers an increased control over locking; it allows a precise specification of resources needed by a routine, and it enables the implementation of interesting scenarios that used to be impossible (or very difficult) to implement. It is now possible to pass around a reference without locking the corresponding object.

One can observe the increased potential for parallelism in the above example: since the handlers of `y` and `z` are not locked, other clients may use them while `r` is being executed. There is no harmful interference between these clients and the processor executing `r` because the latter never performs any calls on `y` or `z`. Therefore, the increased amount of parallelism does not compromise the safety guarantees.

The relaxed locking policy turns out to solve yet another problem of SCOOP\_97: the unclear semantics of void actual arguments passed to a routine expecting separate formals. Should such arguments be locked? If yes, how? A void argument represents no object, therefore there is no handler to be locked. So maybe they should be seen as erroneous? But sometimes, it is necessary to pass `Void` as argument! With the new semantics of attached types, the problem disappears: only detachable arguments can be void but they are not locked anyway.

## 2.2. Support for inheritance and polymorphism

To be usable in practice, the refined locking mechanism must be compatible with inheritance, polymorphism, and dynamic binding. Clients must not be deceived in the presence of polymorphic calls, i.e. the safety guarantees should be preserved even if a redefined version of a feature is applied instead of the original version assumed by the client. In Eiffel, covariant redefinition of argument types is allowed, provided that the redefined arguments are declared as detachable. The rule for result types is less strict: the redefined type has to conform to the original one but does not need to be marked as detachable [ECM05, Rule 8.14.4 /VNCS/].

Indeed, redefinition of a result type from detachable to attached does not cause any problems: if a client assumes the result to be detachable, providing an attached result simply gives a stronger guarantee. Similarly, redefinition of a formal argument from attached to detachable is safe. The client has to use an attached actual argument as required by the original signature; the redefined feature may choose to expect less and only require a detachable argument. The argument is not locked — although the signature of the original feature suggests it — but this causes no harm for the client. On the contrary: the amount of locking is reduced, so the client needs to wait less than with the original feature. (Strictly speaking, the amount of locking is *not higher* than in the original feature.)

**Definition 3 (Feature redefinition rule (tentative)).** The return type of a feature may be redefined in a descendant from detachable to attached. The type of a formal argument may be redefined from attached to detachable.

The routine `r` in Figure 2 is a valid redefinition of the original routine from Figure 1 because the original version takes two arguments of type `separate Y` and `separate Z` whereas the redefined version takes arguments of type `?separate Y` and `?separate Y` respectively; this is consistent with Definition 3.

---

```

r (x: separate X; y: ?separate Y; z: ?separate Z)
  require else
    new_precondition
  do
    if new_precondition then
      -- do something here
    else
      Precursor (x, y, z) -- Invalid
    end
  end

```

---

**Fig. 3.** Problem with `Precursor` calls

The above rule seems to capture the necessary requirements for safe redefinition of features. There are, however, two outstanding problems:

- The use of `Precursor` calls: calls to the inherited version of a feature are not always valid in the redefined body.
- The inherited precondition and postcondition clauses that involve calls on the redefined arguments may become invalid.

Consider the feature `r` in Figure 3 to be a redefined version of `r` from Figure 1. The redefined version lists the precondition `new_precondition`. This weakens the requirements put on clients: the precondition is understood as `some_precondition` or `else new_precondition`. The body of `r` follows a simple pattern: if `new_precondition` holds, some particular actions are taken; otherwise, the original version is called through `Precursor (x, y, z)`. But the precursor call is rejected by the compiler because the types of actual arguments `y` and `z` (`?separate Y` and `?separate Z`) do not conform to the types of the corresponding formals (`separate Y` and `separate Z` respectively) in the original feature. To perform a call to `Precursor`, explicit downcasts (object tests) should be applied to `y` and `z`, as illustrated in Figure 4 (see [Nie07, Chapter 6] for a detailed discussion of the object test mechanism in SCOOP).

While the problem of invalid precursor calls is easy to detect (it amounts to a simple type-check) and to deal with, the problem of contract inheritance is trickier. Consider again the programming pattern used in Figure 3. The `else` branch is taken if `some_precondition` holds (because we know that `new_precondition` is false and `some_precondition` or `else new_precondition` holds); but this assumption is only valid if `some_precondition` does not involve calls on `y` or `z`. What happens if such calls do appear in `some_precondition`? For example, take `some_precondition` to be `x.is_empty` and `y.is_empty`. What is the meaning of `y.is_empty` in the context where `y` becomes detachable? The call `y.is_empty` is valid only if the type of `y` is attached, which obviously is not the case in the redefined version of `r`. Nevertheless, in the context of the original routine where `y` was attached, it was a valid call. It seems that, due to the redefinition of formal arguments from attached to detachable, it is possible to invalidate inherited assertions that involve calls on redefined arguments. There are two alternative ways to prevent it:

1. Assume that all inherited assertions involving calls on detachable formal arguments hold vacuously. For example, if `y` is detachable, `x.is_empty` and `y.is_empty` simply reduces to `True`.
2. Prohibit the redefinition of formal arguments appearing as targets of feature calls in preconditions or postconditions.

The first solution is compatible with the DbC rule for preconditions: inherited preconditions may be weakened. Unfortunately, it is unsound since it generally weakens the postcondition (even if one requires the new postcondition to imply the original one only for pre-states that satisfy the old precondition [PHM99, Par05]). The second solution does not suffer from that drawback. However, it forces programmers to preserve the attached type of a formal argument even if the redefined version of the routine does not rely on any properties of that argument anymore. It might have no importance in the sequential context but in a concurrent context, where detachability implies less locking, such a restriction is burdensome. Essentially, once a formal argument has been used in a precondition or a postcondition, it cannot be redefined from attached to detachable in descendants; there is no possibility to reduce the locking requirements of the routine.

---

```

r (x: separate X; y: ?separate Y; z: ?separate Z)
  require else
    new_precondition
  do
    if new_precondition then
      -- do something here
    elseif {aux_y: separate Y}y and then {aux_z: separate Z}z then
      Precursor (x, aux_y, aux_z) -- Valid
    end
  end

```

---

**Fig. 4.** Correct use of `Precursor`

In practice, we may expect that an attached separate formal argument involved in a postcondition will never be redefined into a detachable one, simply because all redefined versions of a routine have to satisfy at least the original postcondition: there is no way to satisfy that postcondition without the guarantee that no other clients change the state of the object represented by the formal argument. Such a guarantee may only be obtained by locking the argument for the duration of the call, which requires the argument to be declared as attached. On the other hand, a routine that does not lock the given formal argument and needs no assumptions about its state, may simply ignore the precondition clauses concerning that argument, i.e. assume them to be trivially true. Therefore, the two solutions presented above can be combined to yield the refined Feature Redefinition Rule below.

**Definition 4 (Feature redefinition rule (refined)).** The return type of a feature may be redefined from detachable to attached. The type of a formal argument may be redefined from attached to detachable, provided that no calls on that argument appear in the inherited postcondition.

The Inherited Precondition Rule (Definition 5) clarifies the meaning of inherited precondition clauses.

**Definition 5 (Inherited precondition rule).** Inherited precondition clauses involving calls on a detachable formal argument hold vacuously.

This rule puts a higher burden on the redefined version of a routine: if the inherited precondition involves calls on an argument that has been redefined into detachable, the new routine body can make weaker assumptions but must give the same (or stronger) guarantees.

### 2.3. Discussion

Besides the solution presented here, we considered two alternative ways of specifying which formal arguments should be locked. The first option is a compiler optimisation based on the *Business Card Principle* of SCOOP\_97 [Mey97]: if the body of `r` does not perform any calls on `x`, then the processor handling `x` does not need to be locked before `r` is executed. This is decided by the compiler; programmers need no additional type annotations. Unfortunately, this solution is not acceptable for at least three reasons:

- Programmers have no control over locking: the locking behaviour depends on the actual version of the routine chosen at run time.
- Without looking at the implementation of the feature, a client cannot see whether a formal argument is locked or not; the interface is not precise enough to infer all the necessary information. This violates the principle of information hiding.
- The client might be deceived because a redefined version of the feature may lock an argument that the original version does not lock.

The second alternative relies on the extensive use of preconditions. To lock the processor handling a formal argument `x`, an assertion of the form `is_reserved (x)` or — in a more object-oriented style — `x.is_reserved`, must appear in the precondition clause of the enclosing routine.

```
r (x: separate X; y: separate Y; z: separate Z)
  require
    is_reserved (x)
```

Such assertions force the processor executing `r` to block until the corresponding formal arguments are reserved on behalf of that processor. This solution is compatible with polymorphism and dynamic binding: removing `is_reserved (x)` from the redefined precondition eliminates the lock requirement on `x`. Nevertheless, the “locking” part of the redefined precondition should shadow the original one rather than being `or-ed` with it; this obfuscates the resulting precondition. Also, this solution is too verbose; it is much easier to read and write crisp code like

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

than clumsy code like

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

In summary, attached types provide a sound solution which also integrates best with other object-oriented mechanisms and is easy to grasp and apply in practice.

### 3. Lock passing

The next refinement of the access control policy is to make it possible for clients to relinquish their locks temporarily and pass them to a supplier. The mechanism presented here relies on the selective locking introduced in section 2: clients use attached and detachable types to decide whether lock passing should take place. The proposed mechanism allows the implementation of interesting synchronisation scenarios, e.g. separate callbacks, without compromising the atomicity guarantees. We generalise the semantics of argument passing in a way that accommodates the lock passing mechanism and ensures the soundness of the proof technique for SCOOP programs developed in [Nie07, NMO08].

#### 3.1. Need for lock passing

In `SCOOP_97`, a routine holds exclusive locks on its separate suppliers (which have to be formal arguments of the routine) during the execution of its body. This policy ensures that, between two consecutive calls issued by a client, no other client can jump in and modify the state of the supplier object. Such a guarantee is convenient for reasoning about concurrent software but it may limit unnecessarily the expressiveness of the model and increase the danger of deadlocks. Figure 5 illustrates a typical problem caused by cross-client locking. The calls `x.f`, `x.g`, and `y.f` are asynchronous because `f` and `g` are commands so the client does not wait for their completion. Following the *Wait by Necessity* principle [Car93], the client only waits for the result of the query call `x.some_query`. Unfortunately, this causes a deadlock because `x`'s handler is not able to evaluate `some_query` before finishing all the previously requested calls on `x`; one of these calls, `x.g (y)`, needs a lock on `y`'s handler, currently held by the client. But the client cannot unlock `y` before finishing `r`'s body. So, the client is waiting for `x`'s handler and vice-versa; none of them will ever make any progress. (This anomaly was first identified by Phil Brooke and later addressed in a semantic study of `SCOOP_97` [BPJ07] using a different lock-passing approach; see Section 5 for a comparison.)

In fact, getting into a deadlock situation is even simpler; no cross-client locking is necessary. It suffices to

---

```

r (x: separate X; y: separate Y)
  do
    x.f
    x.g (y) -- x waits for y to become available.
    y.f
    ...
    z := x.some_query -- Current waits for some_query to finish.
                    -- DEADLOCK!
  end

```

---

**Fig. 5.** Deadlock caused by cross-client locking

---

```

s (x: separate X)
  do
    z := x.h (Current) -- x waits for Current; Current waits for x to finish.
                    -- DEADLOCK!
  end

```

---

**Fig. 6.** Deadlock caused by a callback

pass **Current** as actual argument of a separate query call, as illustrated in Figure 6. Since feature **h** called on **x** needs to lock the processor that handles **Current**, it will block until that processor can be reserved. But it will never be the case because the client is waiting for the completion of **h**; hence the client’s handler is not idle and cannot be reserved. Again, we have a deadlock; this time, it is caused by a callback (or rather a “lock-back”) of **x**’s handler on **Current**’s handler. The body of **h** does not even need to perform any real callback to cause a deadlock!

Meyer [Mey97] suggests solving the callback problem by applying the *Business Card Principle*, discussed earlier, which stipulates that clients may only pass a reference to **Current** to features that do not lock the corresponding formal argument, i.e. whose body does not contain any calls on that argument. Unfortunately, the principle actually prohibits separate callbacks rather than handling them.

### 3.2. Mechanism

In the two examples above, a deadlock occurs at the moment when the client waits for one of its suppliers. Since the client is waiting, it does not perform any operations. Therefore, it makes no use of the locks it holds. If the client could temporarily pass the lock on **y** (in Figure 5) respectively on **Current** (in Figure 6) to its supplier **x**, the supplier would be able to execute the requested feature, return the result, and let the client continue, thus avoiding the deadlock. This basic idea is simple but, besides solving the problem of cross-client locking and separate callbacks, the lock passing mechanism has to satisfy additional requirements:

- *It must not compromise the atomicity guarantees.*  
Sound reasoning about feature calls is only possible if other clients do not interfere, i.e. the accesses to a given object are atomic. This immediately rules out a solution whereby a client passes a lock on an object to a supplier and then continues its own execution: it would be impossible to decide statically how the client’s and the supplier’s calls on the locked object are ordered. As a result, neither the client nor the supplier would be able to ensure the correctness of their calls. Additionally, assertions involving calls on the concerned object would not be usable.
- *Clients must be able to decide whether to pass or not to pass a lock.*  
It must be clear from the program text whether lock passing occurs. The mechanism should be controlled by clients, i.e. they should have the choice to pass or not to pass a lock to a supplier that needs it. Letting a supplier snatch a lock without asking for the client’s permission is unacceptable; it complicates the reasoning about programs and it may lead to an arbitrary interleaving of accesses to the locked object, thus compromising the atomicity.

---

```

-- in class C
r (x: separate X; y: separate Y)
do
  x.f
  x.g (y) -- Current passes its locks to x
           -- and waits until g terminates.
  y.f
  ...
  z := x.some_query -- No deadlock here
end

-- in class X
g (y: separate Y)
do
  ...
end

```

---

**Fig. 7.** Cross-client locking without deadlock

- *The mechanism should increase the expressiveness of the language, not restrict it.*  
Lock passing should enable the implementation of additional interesting synchronisation scenarios not supported in the basic model. On the other hand, all scenarios implementable in the basic model should be expressible in the extended framework as well.
- *The solution must be simple and well integrated with other language mechanisms.*  
The solution must be sound in the presence of polymorphism and dynamic binding; it has to be compatible with the rules of DbC as well.

We propose the following solution: if a feature call  $x.f(a_1, \dots, a_n)$  occurs in the context of the routine  $r$  where some actual argument  $a_i$  is *controlled*, i.e.  $a_i$  is attached and locked by  $r$  (see Definition 6 below), and the corresponding formal argument of  $f$  is declared as attached, the client's handler (the processor executing  $r$ ) passes all currently held locks (including the implicit lock on itself) to the handler of  $x$ , and waits until  $f$  has terminated. When the execution of  $f$  is complete, the client's handler gets back the locks and resumes the computation.

**Definition 6 (Controlled expression).** An expression  $\text{exp}$  is controlled if and only if  $\text{exp}$  is attached, i.e. statically known to be non-void, and either (a)  $\text{exp}$  is non-separate, or (b)  $\text{exp}$  is handled by the same processor as some formal argument of the routine  $r$  in which  $\text{exp}$  appears.

Let's see how our mechanism solves the problems of cross-client locking and separate callbacks. Feature  $r$  in Figure 7 is identical with feature  $r$  from Figure 5 but, thanks to lock passing, it does not deadlock: the call  $x.g(y)$  is executed synchronously, with the client passing on all its locks to  $x$  for the duration of  $g$ . No deadlock occurs when the client evaluates  $x.some\_query$  because the handler of  $x$  is not blocked anymore; the execution of  $x.g(y)$  has terminated so  $x.some\_query$  can be applied. Similarly, the routine  $s$  in Figure 8 does not deadlock anymore because  $x.h(\text{Current})$  results in the lock passing which lets  $x$ 's handler obtain a lock on  $\text{Current}$  without waiting. (In this particular case, the client and the actual argument are both handled by the same processor; we assume that every processor, when non-idle, implicitly holds a lock on itself.)

That last example raises an interesting issue: if the body of  $h$  indeed performs a callback, i.e. a call on a target handled by the processor that has passed its locks, how should such a call be treated? Consider the call  $c.f(\dots)$  in Figure 8; does it have the usual asynchronous semantics whereby a request to execute  $f$  is queued on  $c$ 's handler? If yes, then the problem of deadlock is not really solved but just postponed. To avoid this,  $c.f(\dots)$  should be performed synchronously, i.e. scheduled it for an immediate execution, so that  $c$ 's handler — the one that has initially passed its locks and is waiting for the termination of  $x.h(\text{Current})$  — has a chance to execute it. So, the call is separate but synchronous; this may seem a bit disturbing. A

---

```

-- in class C
s (x: separate X)
do
  z := x.h (Current) -- x gets lock on Current.
                      -- No deadlock here
end

r (x: separate X)
do
  x.g (...)
  ...
end

-- in class X
h (c: separate C): Z
do
  c.f (...)
  c.r (Current)
end

```

---

**Fig. 8.** Callback without deadlock

closer inspection, however, reveals the underlying reason for applying this semantics: the target is handled by a processor that holds a lock on the current processor. This is just like for non-separate calls, where the target's handler — which happens to be the current processor itself — holds a lock on the current processor. Therefore, we can generalise the applicability of the synchronous call semantics to all the calls whose target's handler holds a lock on the current processor. (All such calls may be viewed as separate callbacks.) Note that this rule permits nested (or even recursive) lock passing, i.e. the body of `h` could involve calls resulting in lock passing whereby the locks obtained at the previous step are passed further; those features could involve calls with lock passing, and so on. No limit on the depth of lock passing is imposed; the atomicity guarantees are preserved because the client always blocks. For instance, the call `c.r (Current)` in the body of `h` causes locks to be passed from the handler of `x` to the handler of `c` (which happens to have passed its locks in the previous step) and then may be passed again in the opposite direction as a result of some call on `x` in the body of `r`. When that innermost callback terminates, the locks passed at that level are revoked from the handler of `x`, permitting the call `c.r (Current)` to return, which involves revoking the locks from the handler of `c`; in consequence, the body of `h` and the call `x.h (Current)` terminate, which results in the revocation of the locks passed originally to the handler of `x`.

Two more points need to be clarified. Firstly, whenever the lock passing occurs, the client passes *all* its locks to the supplier, not only the locks corresponding to the particular arguments that triggered the mechanism. Such a generous behaviour of clients eliminates more potential deadlocks than passing just the specified locks. The client does not use any locks anyway while it is blocked so it does not hurt to pass locks “just in case”. On the contrary, the supplier might make use of these additional locks in the body of the requested routine that perhaps would deadlock otherwise. Secondly, the same processor may appear several times in a chain of lock passing, e.g. if  $P_x$  passes its locks to  $P_y$  which then passes its locks to  $P_z$  which then passes its locks to  $P_x$ , then  $P_x$  holds all the locks it originally held plus any additional locks acquired by  $P_y$  and  $P_z$  underway. Naturally, passing the locks back occurs in the reverse order:  $P_x$  to  $P_z$  to  $P_y$  to  $P_x$ .

Definition 7 captures the new semantics of the feature call mechanism, reflecting the refined meaning of argument passing and the additional synchrony requirement; it refines the feature call semantics proposed in [Nie07, Section 6.1].

**Definition 7 (Feature call semantics (refined)).** A feature call `x.f ( $\bar{a}$ )` results in the following sequence of actions performed by the client's handler  $P_c$ :

1. Argument passing: bind the formal arguments of `f` to the corresponding actual arguments  $\bar{a}$ . If any attached formal argument corresponds to a controlled actual argument of a reference type, pass all the currently held locks (including a lock on  $P_c$ ) to the supplier's handler  $P_x$ .

	formal attached	formal detachable
actual (ref) controlled	<b>yes</b>	<i>no</i>
actual (ref) uncontrolled	<i>no</i>	<i>no</i>
actual expanded	<i>no</i>	<i>no</i>

**Fig. 9.** Lock passing combinations

2. Feature request: ask  $P_x$  to apply  $\mathbf{f}$  to  $\mathbf{x}$ .
  - (a) Schedule  $\mathbf{f}$  for an immediate execution by  $P_x$  and wait until it terminates, if any of the following conditions holds:
    - The call is non-separate, i.e.  $P_c = P_x$ .
    - The call is a separate callback, i.e.  $P_x$  has previously passed its locks (directly or indirectly) to  $P_c$ . (In other words:  $P_x$  precedes  $P_c$  in a chain of lock passing.)
  - (b) Otherwise, schedule  $\mathbf{f}$  to execute after the previous calls on  $P_x$ .
3. Wait by necessity: if  $\mathbf{f}$  is a query, wait for its result.
4. Lock revocation: if lock passing occurred in step 1, wait for  $\mathbf{f}$  to terminate, then revoke the locks from  $P_x$ .

### 3.3. Lock passing in practice

Figure 9 recapitulates the possible type combinations of formal and actual arguments, and the resulting semantics of argument passing (*yes* stands for “lock passing takes place”, *no* stands for “no lock passing”). The example in Figure 10 illustrates the possible combinations of separate and non-separate calls with and without wait by necessity and lock passing. The current object (an instance of  $\mathcal{C}$ ) is handled by the processor  $P_c$ , and  $\mathbf{x}$ ,  $\mathbf{y}$ ,  $\mathbf{my\_c}$ , and  $\mathbf{my\_z}$  are handled by (different) processors  $P_x$ ,  $P_y$ ,  $P_{\mathbf{my\_c}}$ , and  $P_{\mathbf{my\_z}}$  respectively. The calls appearing in the body of  $\mathbf{r}$  have the following semantics:

- (Command call  $\mathbf{my\_x.f}$  (5)) The call is non-separate because  $\mathbf{my\_x}$  is handled by  $P_c$ ; the current execution state is saved on  $P_c$ ’s call stack, and  $\mathbf{my\_x.f}$  (5) is executed synchronously. No lock passing occurs because the actual argument is expanded.
- (Command call  $\mathbf{my\_x.g}$  ( $\mathbf{x}$ )) Similar to the previous call but lock passing occurs. However, it is vacuous because both the client and the supplier objects are handled by  $P_c$ .
- (Query call  $\mathbf{my\_x.h}$  (**Current**)) Similar to the previous call but wait by necessity applies. Lock passing occurs but is vacuous.
- (Command call  $\mathbf{x.f}$  (10)) The call is separate because  $P_c \neq P_x$ . No lock passing occurs because the actual argument is expanded.
- (Command call  $\mathbf{x.g}$  ( $\mathbf{my\_z}$ )) Similar to the previous call. No lock passing occurs because the actual argument  $\mathbf{my\_z}$  is not controlled.
- (Command call  $\mathbf{x.g}$  ( $\mathbf{y}$ )) Similar to the previous call but lock passing occurs because the actual argument  $\mathbf{y}$  is controlled and the corresponding formal is attached.  $P_c$  cannot move to the next operation until  $P_x$  has terminated the application of  $\mathbf{x.g}$  ( $\mathbf{y}$ ), i.e. after servicing all the previous calls in its request queue.
- (Command call  $\mathbf{x.m}$  ( $\mathbf{y}$ )) No lock passing occurs because the formal argument  $\mathbf{a}$  is detachable.
- (Query call  $\mathbf{x.h}$  ( $\mathbf{my\_c}$ )) Wait by necessity applies. No lock passing occurs because the actual argument  $\mathbf{my\_c}$  is not controlled.
- (Query call  $\mathbf{x.h}$  (**Current**)) Similar to the previous call but lock passing occurs because the actual argument **Current** is controlled.

That last call is particularly interesting because it involves a separate callback: during the application of  $\mathbf{x.h}$  (**Current**),  $P_x$  performs the call  $\mathbf{c.f}$  (...) where  $\mathbf{c}$  is the actual argument of the call  $\mathbf{x.h}$  (**Current**).

---

```

-- in class C
my_x: X
my_z: separate Z
my_c: separate C
i: INTEGER

r (x: separate X; y: separate Y)
  do
    my_x.f (5) -- non-separate, no lock passing
    my_x.g (x) -- non-separate, lock passing (vacuous)
    i := my_x.h (Current) -- non-separate, lock passing (vacuous)

    x.f (10) -- separate, no wait by necessity, no lock passing
    x.g (my_z) -- separate, no wait by necessity, no lock passing
    x.g (y) -- separate, no wait by necessity, lock passing
    x.m (y) -- separate, no wait by necessity, no lock passing
    i := x.h (my_c) -- separate, wait by necessity, no lock passing
    i := x.h (Current) -- separate, wait by necessity, lock passing
  end

-- in class X
f (i: INTEGER) do ... end

g (a: separate ANY) do ... end

h (c: separate C): Z do c.f (...) end

m (a: ?separate ANY) do ... end

```

---

**Fig. 10.** Lock passing example

The target of the call `c.f (...)` is handled by  $P_c$ , and  $P_c$  precedes  $P_x$  in the chain of lock-passing (it has just passed its locks to  $P_x$ ). Therefore, the call is handled as a separate callback (according to step 2a in Definition 7), i.e.  $P_c$  immediately executes the feature requested by  $P_x$ ; the latter waits until the feature has terminated. It is important to observe that the request queue of  $P_c$  is not involved in handling this request; the feature is handled using the call stack, just like a non-separate call. That is why  $P_c$ 's queue remains empty throughout the execution of `x.h (Current)`. (For a detailed discussion on how lock passing is implemented in practice, see [Nie07, Section 11.2.5].)

The lock passing mechanism influences the semantics of feature calls so that certain calls, e.g. `x.g (y)` in Figure 10, have a different meaning in SCOOP than in SCOOP\_97. Nevertheless, it is possible to emulate the original semantics — at a cost of a few additional lines of code — as illustrated in Figure 11. The original feature `g` from Figure 10 has been replaced by a pair of features: `g` and `blocking_g`. The formal argument of `g` is now detachable, so the call `x.g (y)` does not involve lock passing, even though the actual argument `y` is controlled. The call to the auxiliary feature `blocking_g` will later lock `y` but it does not influence the semantics of `x.g (y)` as seen by the client; the call `x.g (y)` is non-blocking, just as it would be in SCOOP\_97. An object test is used in the body of `g` for downcasting a detachable type to an attached type.

### 3.4. Discussion

The proposed mechanism fulfils all the additional requirements discussed at the beginning of this section. First, it does not compromise the atomicity because locks are passed to the supplier's handler only for the duration of `f`; since the client is blocked in the meantime, there is no danger of harmful interleaving with other clients. Of course, the supplier is free to perform any sequence of calls on the locked objects but the client

---

```

-- in class C
r (x: separate X; y: separate Y)
  do
    ...
    x.g (y) -- No lock passing here.
    ...
  end

-- in class X
g (y: ?separate Y)
  local
    aux_y: separate Y
  do
    if {aux_y: separate Y} y then
      blocking_g (aux_y)
    end
  end
end

blocking_g (y: separate Y)
  do
    ... -- body of original g
  end

```

---

Fig. 11. Emulating SCOOP\_97 semantics

knows that all these calls will be executed before its own subsequent calls; additionally, the postcondition of  $f$  stipulates what the supplier may or may not do with these objects. Second, it is clear from the program text whether a lock passing occurs: the controllability of actual arguments is immediately deducible from their type; the type of the formal arguments of  $f$  is known from  $f$ 's signature. Therefore, a client can decide to pass or not to pass its locks simply by using controlled or uncontrolled actual arguments; alternatively, it may use a feature that takes detachable formals. Note the absence of lock passing for actual arguments of an expanded type (even though they are always controlled). This reflects the copy semantics of such arguments: the corresponding formals are bound to copies of actuals. Since these copies are non-separate from the supplier, no lock passing is necessary to give the supplier the control over them.

Lock passing increases the expressiveness of the programming framework: several scenarios not supported by SCOOP\_97 — including the cross-client locking and separate callbacks illustrated in Figures 7 and 8 — are now implementable. Section 3.3 demonstrates that the locking policy of SCOOP\_97 can be simply emulated in SCOOP; therefore, the backward compatibility is preserved and the mechanism does not limit the expressive power of our framework.

#### 4. Impact of lock passing on formal reasoning

The proof technique for SCOOP programs introduced in [Nie07, NMO08] supports sequential-like reasoning about concurrent code involving asynchronous calls. Essentially, the proof technique follows the traditional approach to reasoning about feature calls whereby suppliers assume the precondition on entry to the routine body and must establish the postcondition on exit, whereas clients must establish the precondition before the call and assume the postcondition after the call. However, only *controlled* precondition and postcondition clauses, i.e. assertions that only involve calls on targets controlled by the client (following Definition 6), are considered. The proof technique assumes the lock passing mechanism proposed here: indeed, reasoning would be unsound without it. Consider the proof sketch in Figure 12. The call  $x.transfer\_to(y)$  is processed synchronously because  $y$  is controlled, so the lock passing occurs. Any calls on  $y$  within the body of  $transfer\_to$  are guaranteed to be executed before the subsequent call to  $y.empty$  issued by the client. Therefore, the postcondition of  $transfer\_to$  may be assumed before the call  $y.empty$ ; it is necessary to prove the correctness of that call (and the whole routine  $s$ ). Without the lock passing mechanism, the call

---

```

-- in class C
s (x, y: separate X)
  require
    x.is_empty and y.is_empty
  do
    -- {x.is_empty and y.is_empty}
    x.fill
    -- {x.is_full and y.is_empty}
    x.transfer_to (y)
    -- {y.is_full}
    y.empty
    -- {y.is_empty}
  ensure
    y.is_empty
  end

-- in class X
fill
  -- Fill the container.
  require
    is_empty
  ensure
    is_full

empty
  -- Empty the container.
  require
    is_full
  ensure
    is_empty

transfer_to (y: separate X)
  -- Transfer the contents to y.
  require
    y.is_empty
  ensure
    y.is_full

```

---

**Fig. 12.** Sequential-like reasoning about concurrent code using lock passing

`y.empty` could be processed before the calls issued by the body of `transfer_to`; the assumption `y.is_full` would be false and the call `y.empty` invalid. In fact, following SCOOP\_97 rules, `x` would be able to execute `transfer_to` only after the client terminated `s` and unlocked `y`; therefore, `y.is_full` would eventually become true, which contradicts the promise made by `s`: its postcondition says that `y` is empty!

## 5. Related work

This article extends our previous work on the access control policy for SCOOP [Nie06a] which discussed the use of detachable types but did not cover the problems related to inheritance and polymorphism; lock passing was only described shortly, without considering the complex scenarios discussed here. A basic mechanism for shared locking, based on a refined notion of a *pure query* and a new semantics for the `only` clauses (used in the sequential Eiffel to express the frame properties of features), was presented in [Nie06b]. The mechanism proved unsound in the presence of polymorphism, therefore we do not consider it here. We are currently refining it to support inheritance and polymorphism.

---

```

-- in class C
r (x: separate X)
  do
    s (x) -- Lock on x is passed from r to s.
    ...
    y := x.g -- Although y is non-separate from x,
    s (y) -- this call may still block.
    ...
  end

-- in class X
g: X -- Result is non-separate.

```

---

**Fig. 13.** Alternative lock passing model [BPJ07]

Meyer [Mey05] discusses the attached type mechanism, in particular its use for eliminating *catcalls*. He also describes the possible application of attached types to concurrency. The problem of contract redefinition in a concurrent context is not addressed but the article prompted us to have a closer look at the contract inheritance mechanism. Meyer’s rule for argument redefinition requires a covariantly redefined type of a formal argument to be marked as detachable [ECM05, Rule 8.14.4 /VNCS/]. Inherited assertions involving calls on detachable arguments are evaluated using an implicit object test. For example, for attached  $x$  and detachable  $y$ , the expression

```
x.is_empty and y.is_empty
```

is understood as

```
x.is_empty and ({aux_y: Y}y implies aux_y.is_empty)
```

hence  $x.is\_empty$  if  $y$  is void. Besides being complicated, this solution is inconsistent with DbC: as demonstrated in section 2.2, it may lead to postcondition weakening. Our Rules 4 and 5 solve this problem: they may be combined with the Eiffel rule to ensure the consistency with DbC and to simplify the treatment of inherited contracts. Although initially developed to address a concurrency issue, our technique proves useful in a sequential context as well.

Brooke et al. [BPJ07] discuss lock passing as part of their CSP semantics for SCOOP. There are a number of differences with respect to our model. First of all, locks are held by calls rather than processors, and passed from one call to another, e.g. if a call  $s(x)$  occurs in the body of  $r$ , and  $r$  holds a lock on  $x$ , then this lock is passed to  $s$  (see Figure 13). The separateness of a call’s target is irrelevant to lock passing: separate and non-separate calls are treated alike. This simplifies the CSP semantics considerably but makes the model less practical: the authors demonstrate that, depending on the overall system configuration, a chain of non-separate calls may lead to deadlock if indirect lock passing is disallowed; the problem may occur even with unqualified calls, i.e. calls whose target is **Current**. (In our model, non-separate calls cannot cause such problems because the client and the supplier are handled by the same processor, so the supplier has the same set of locks as the client; this behaviour is similar to the indirect lock passing variant in [BPJ07].) Also, only simple callback scenarios are supported; recursive callbacks may result in deadlocks when combined with synchronous calls. (Our model permits recursive callbacks without deadlocking by passing all locks and making these calls synchronous.) Finally, since locks protect single objects rather than whole processors, it is possible for a call to block waiting for a lock on an object even though the enclosing call already holds a lock on another object that is non-separate from the former. Figure 13 gives an example:  $s(y)$  may block although it occurs in the body of  $r$  already holding a lock on  $x$  which is non-separate from  $y$ .

The solution proposed in [BPJ07] has an advantage: the potential for parallelism is increased because clients do not necessarily wait for termination of calls involving lock passing as in our model. This means, however, that assertional reasoning about concurrent code becomes more complicated, for the reasons outlined in section 4.

## 6. Conclusions

We have presented two simple refinements of the access control policy for SCOOP. The selective locking mechanism, based on the new semantics of attached and detachable types, supports precise specification of locking requirements of routines, thus eliminating unnecessary synchronisation often exhibited in SCOOP\_97 programs. The lock passing mechanism permits passing locks from clients to suppliers for the duration of a single call. Both mechanisms greatly improve the flexibility of the model by exploiting the potential parallelism and reducing the danger of deadlocks, while preserving strong atomicity guarantees offered by SCOOP. The proposed solution allows implementation of many synchronisation scenarios that were difficult (or impossible) to express in the original model; at the same time, all scenarios implementable in the original model can also be expressed in the extended framework. Both mechanisms support polymorphism and dynamic binding; they are compatible with the general rules of DbC, and underpin the novel proof technique for concurrent programs described in [NMO08].

We have implemented the flexible access control policy in the *scoop2scoopli* compiler and the supporting *SCOOPLI* library [Nie07]. These tools are now integrated with the EiffelStudio IDE and available for download at <http://se.ethz.ch/research/scoop.html>. A soundness proof of the proposed mechanisms is part of the ongoing work on the formal framework for SCOOP.

## 7. Acknowledgements

Bertrand Meyer contributed largely to the development of the attached type mechanism and suggested its possible application in the context of SCOOP. Bernd Schoeller pointed out the problem of precursor calls in redefined features with modified types of formal arguments. We are grateful to Phil Brooke, Richard Paige, Jonathan Ostroff, and Hai Feng Huang for their extensive feedback on the proposed mechanisms. This research work was conducted as part of the author's PhD studies at the ETH Zurich, with the support of the Hasler Foundation, Berne.

## References

- [BPJ07] Phillip J. Brooke, Richard F. Paige, and Jeremy L. Jacob. A CSP model of Eiffel's SCOOP. *Formal Aspects of Computing*, 19(4):487–512, 2007.
- [Car93] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.
- [ECM05] ECMA. *ECMA-367: Eiffel analysis, design and programming language*. European Association for Standardizing Information and Communication Systems, June 2005.
- [FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 302–312, 2003.
- [ISO06] ISO. *ISO/IEC DIS 25436: Eiffel analysis, design and programming language*. International Organization for Standardization, June 2006.
- [Mey92] Bertrand Meyer. Applying “Design by Contract”. *IEEE Computer*, 25(10):40–51, October 1992.
- [Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [Mey05] Bertrand Meyer. Attached types and their application to three open problems of object-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 1–32, July 2005.
- [Nie06a] Piotr Nienaltowski. Flexible locking in SCOOP. In *International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, pages 71–90, York, United Kingdom, July 2006.
- [Nie06b] Piotr Nienaltowski. Refined access control policy for SCOOP. Technical Report 511, Computer Science Department, ETH Zurich, February 2006.
- [Nie07] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, no. 17061, Department of Computer Science, ETH Zurich, 2007.
- [NMO08] Piotr Nienaltowski, Bertrand Meyer, and Jonathan S. Ostroff. Contracts for concurrency. *Formal Aspects of Computing*, DOI 10.1007/s00165-007-0063-2, 2008.
- [Par05] Matthew J. Parkinson. *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, UK, 2005.
- [PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. volume LNCS 1576, pages 162–176, 1999.