Flexible locking in 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. This results in increased potential for deadlocks; additionally, some interesting synchronisation scenarios cannot be implemented efficiently. This paper presents two mechanisms that increase the flexibility of locking in SCOOP: (1) a type-based mechanism to specify which arguments of a routine call should be locked, and (2) a lock-passing mechanism that allows for safe handling of callbacks and complex synchronisation scenarios that involve mutual locking of several separate objects. When combined, these two approaches greatly increase the expressive power of SCOOP and reduce the risk of deadlock.

1 Introduction

Controlling access to shared resources is one of the main problems in concurrent programming. Uncontrolled access to shared resources is very dangerous as it may lead to an inconsistent program state. In procedural programming, solutions to conflict problems involve proper synchronisation among processes based on the concept of critical section – a process requesting a shared resource has to wait for executing its critical section if another process is currently accessing the shared resource. 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 may 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. Also, all separate arguments of a feature call have to

be locked, even if they are never used by the feature. Additionally, a client that holds a lock on a given resource cannot relinquish it temporarily when the lock is not needed. As a result, certain scenarios, e.g. callbacks involving separate suppliers, cannot be implemented. In most cases, the amount of locking is higher than necessary. Such a pessimistic locking policy makes SCOOP-based programs more deadlock-prone.

We present two 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 clients to temporarily pass on their locks to separate suppliers. We illustrate the discussion with numerous code examples.

The article is organised in the following way. Section 2 shortly describes the basic synchronisation policy of SCOOP. Section 3 describes the type-based mechanism for precise specification of formal arguments to be locked. Problems of precondition weakening and precursor calls discussed in that section are not concurrency-specific; their analysis and the proposed solution (rule 1') may be regarded as contributions to DbC in general. Section 4 introduces the lock-passing mechanism. Section 5 discusses related work. Finally, Section 6 concludes the article and describes future research directions.

The use of detachable and attached types in the context of SCOOP is part of joint work with Bertrand Meyer. 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, whereby suppliers are allowed to "snatch" a lock from their clients when necessary. Although we use the same name for our mechanism, we follow a different approach here: we require a client to pass locks explicitly; our goal is to increase the flexibility of the model while preserving the possibility to reason about the order of feature calls. The differences between both solutions are discussed in section 5.

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.

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)
    do
        ...
        an_x.f (a)
        ...
    end
```

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 above, 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 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. In a separate article [4] we propose a generalised semantics for contracts in SCOOP that unifies the concepts of preconditions and wait-conditions.

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* [5]. The lock-passing mechanism described in section 4 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 [6]; it is fully compatible with other

object-oriented concepts such as polymorphism, inheritance, and genericity. The application of detachable types to SCOOP is a result of joint work with Bertrand Meyer; the basic idea was described in [7]. Here, we take a closer look at the mechanism, discuss its applications, and study its impact on other language features.

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 feature r in Figure 1. According to SCOOP, the processors that handle x, y, and z must be locked by

```
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  -- separate call
        my_y := y
        x.g  -- separate call
        my_z := z
        s (z)
    end
```

Fig. 1. Original feature

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 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 [6]. Every type is declared either as "attached" or as "detachable"; an attached type guar-

antees 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) [7]. 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 attached formal arguments of a routine be locked. Conversely, no detachable formal arguments are locked. This is not a mere overloading of the semantics of detachable types. In fact, this rule captures the essence of call validity: a client is allowed to make a call if and only if the target is non-void and the client has exclusive access to the target's processor. We use attached annotations to satisfy both requirements. Let's apply the rule to the example in Figure 1. 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 (see Figure 2). Note that the applied rule is consistent

```
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. Redefined feature

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 to 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 Rule 1.

Rule 1. Result and argument redefinition.

- The return type of a feature may be redefined from ?T to T.
- The type of a formal argument may be redefined from T to ?T.

If the original version of the feature takes an argument of type **separate** *T*, we can redefine it in a descendant so that it takes an argument of type ?**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 conform to 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 a 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.

There are, however, two problems related to the use of contravariant redefinition:

- The use of **Precursor** calls is not always possible.
- Inherited precondition and postcondition clauses that involve calls on redefined formal arguments may become invalid.

Consider the common programming pattern depicted in Figure 3. The redefined version of feature *r* lists precondition *new_precondition* that weakens the requirements put on clients (assume that the original feature is depicted in Figure 1). The body of *r* follows a simple pattern: if *new_precondition* holds, some particular actions corresponding to that new case are taken; otherwise, **Precursor** (*x*, *y*, *z*) is called. But this call will be rejected by the compiler because the types of actual arguments *y* and *z* (?**separate** *Y* and ?**separate** *Z*, respectively) do not conform to the types of the corresponding formals (**separate** *Y* and **separate** *Z*, respectively). In order to use calls to **Precursor**, explicit downcasts (object tests) must be performed.

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

Fig. 3. Use of **Precursor**

While the problem of invalid precursor calls is easy to detect (it amounts to a simple type-check performed by the compiler) and to deal with, the second problem mentioned above – contract inheritance – is much trickier. Consider again the programming pattern used in Figure 3. The **else** part implicitly assumes that *some_precondition* holds because we know that *some_precondition* **or else** *new_precondition* holds and *new_precondition* is false. This assumption is 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 is of a detachable type? According to the call validity rule, call $y.is\_empty$ is valid only if the type of y is attached, which obviously is not the case here. Nevertheless, in the context of the inherited routine where y was attached, it was a valid call. So, it seems that we have a problem with contract inheritance – due to contravariant redefinition of formal arguments from attached to detachable, it is possible to invalidate inherited assertions that involve calls on redefined arguments. There are two simple solutions to this problem:

1. Ignore all inherited assertions that involve calls on detachable formal arguments, i.e. assume that these assertions hold vacuously. For example, $x.is\_empty$ **and** $y.is\_empty$ would reduce to $x.is\_empty$ **and true** hence to $x.is\_empty$ if y is detachable.
2. Prohibit the redefinition of formal arguments involved as targets of feature calls in preconditions and postconditions.

The first solution is compatible with the rules of Design by Contract when applied to preconditions – inherited preconditions are simply weakened. Unfortunately, postconditions may get weakened too, which is clearly against the rules of DbC. The second solution does not suffer from that drawback. Nevertheless, it forces the programmer 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 the detachability of an argument implies less locking, such restriction is very unwelcome. Essentially, once a formal argument has been used in a precondition or a postcondition, it cannot be redefined from attached to detachable in descendants. This means that there is no possibility to reduce the locking requirements of the routine.

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 the original postcondition (possibly strengthened) and there is no way to satisfy the postcondition without the guarantee that no other clients may change the state of the object represented by the formal argument. Such guarantee may only be obtained by locking the argument for the duration of the call which will only happen if the type of the argument is attached. On the other hand, it is logical that a redefined version of a routine that does not need to lock a given formal argument does not make any assumptions about the state of the object represented by that argument, i.e. it simply ignores the precondition clauses concerning that

argument. Therefore, we could combine both solutions presented above into one solution that is both sound (i.e. it follows the principles of Design by Contract) and flexible. We disallow the redefinition of a formal argument from attached to detachable if the inherited postcondition involves calls on that formal argument. No such restrictions are put on arguments involved in preconditions; if an inherited precondition clause involves a call on a detachable formal argument, that clause is considered to hold vacuously. We refine the rule for result and argument redefinition accordingly.

Rule 1'. Result and argument redefinition (refined).

- The return type of a feature may be redefined from ?*T* to *T*.
- The type of formal argument *x* may be redefined from *T* to ?*T*, provided that no calls on *x* appear in the inherited postcondition.

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 preconditions. 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. The fact that *x* is a formal argument of the routine does not automatically imply locking.

```
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 force clients to 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 **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)
    require
        is_available (x)
        is_available (y)
        is_available (z)
    do ...
    end
```

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 propose it as the standard approach.

4 Lock passing

The next step to refine the access control policy and increase the expressiveness of the model is to allow clients to temporarily pass on their locks to their separate suppliers when needed. This was impossible to implement in the original SCOOP model where clients would keep exclusive locks during the execution of the routine that acquired the locks. Our approach relies on the mechanism described in section 3 – clients and suppliers use detachable and attached types to specify whether lock passing should take place. The proposed mechanism makes concurrent programs less deadlock-prone and allows programmers to implement interesting synchronisation scenarios.

4.1 The need for lock passing

In SCOOP, clients executing a routine that locks separate suppliers hold exclusive locks on these suppliers during the whole duration of the routine call. As pointed out in section 2.3, this policy ensures that no other client can jump in and modify the state of the supplier object between two consecutive calls issued

by our client. While such a guarantee is very convenient for reasoning about concurrent software – we may apply similar techniques as for sequential programs – it unnecessarily limits the expressiveness of SCOOP and leads to deadlocks. To illustrate the problems caused by the restrictive locking policy, we use a simple example in Figure 4. Calls to $x.f$, $x.g$, and $y.f$ are asynchronous (f and g are

```
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 x.
                            -- DEADLOCK!
    end
```

Fig. 4. Deadlock caused by cross-client locking

commands), so the client will not wait for their completion. In fact, following the *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, as in Figure 5. Since feature g called on x needs to lock the processor that handles **Current**, it will block until that processor is unlocked. But it will never be unlocked because it is waiting for the completion of the call to g. Again, we have a deadlock. This time, it is caused by a callback (or rather a "lock-back") of g's processor on **Current**'s processor. Note that the body of g does not even need to involve any real callback on **Current** in order to cause a deadlock.

```
s (x: separate X)
    do
        z := x.g (Current) -- x waits for Current; Current waits for x.
                           -- DEADLOCK!
    end
```

Fig. 5. Deadlock caused by a callback

Meyer [1] suggested that the problem depicted in Figure 5 could be solved by the use of the *business card principle* – clients may only pass the 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 business card principle does not work well with inheritance and polymorphism – it suffers from the same drawbacks as the first alternative approach to locking that we discussed in section 3.4. Also, it only solves the problem if there are no callbacks in the body of routine g. In the presence of actual callbacks, we would still end up with a deadlock.

Note that, in both examples, the 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 on its suppliers. Therefore, it makes no use of the locks it holds. If the client could temporarily pass on the lock on y (in Figure 4) respectively on **Current** (in Figure 5) to its supplier x, the supplier would be able to execute the requested feature and return the result, which would allow the client to continue. We would be able to avoid deadlock. We use that observation to develop a lock passing mechanism that allows clients to agree to "lend" their locks to suppliers for the duration of a single separate call. The solution proposed by Brooke et al. [2] takes the opposite approach – it allows suppliers to get locks from clients without their consent. See section 5 for a comparison of both approaches.

4.2 The mechanism

We cannot simply say that locks are passed whenever possible as this would limit the number of synchronisation scenarios that can be implemented. In particular, some synchronisation scenarios supported by the original model would not be implementable in the extended SCOOP. Obviously, we want to preserve the backward-compatibility with the original model while making it more flexible and expressive. We want to give the programmers the possibility to decide whether lock passing should take place in a given situation or not. Once again, detachable types offer a simple solution. We introduce the lock passing rule based on the new semantics for detachable types and argument passing.

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

Let us re-consider our examples. Feature r in Figure 6 is identical with feature r from Figure 4 but the semantics of argument passing follows Rule 2. As a result, call $x.g\ (y)$ will be executed synchronously, with the client passing on all its locks to x. No deadlock occurs at the moment when the client evaluates $x.some\_query$ because x is not blocked anymore, as it was the case in Figure 4.

```
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
```

Fig. 6. Cross-client locking without deadlock

Similarly, the problem of separate callbacks can be solved thanks to lock passing. Routine s in Figure 7 is not deadlock-prone anymore because separate call $x.g$ (**Current**) results in lock passing that allows x to obtain a lock on **Current** without waiting. In this particular case Rule 2 has been applied taking $P_1 = P_3$ – the client and the actual argument are both **Current**, therefore they are handled by the same processor; we assume that every processor, when non-idle, implicitly holds a lock on itself. Note that, whenever lock passing occurs

```
s (x: separate X)
    do
        z := x.g (Current) -- x gets lock on Current from Current.
                           -- No deadlock here!
    end
```

Fig. 7. Callback without deadlock

as a result of a feature call, the client passes *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 the client does not use any locks anyway while waiting until the execution of the supplier's 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 avoids more potential deadlocks than passing just the specified locks.

4.3 Lock passing in practice

We said that programmers should be able to decide whether lock passing takes place or not, and that the new locking policy should be backwards-compatible with the original SCOOP approach. This means that all scenarios supported by SCOOP should be easily implementable in the extended model.

This flexibility can be achieved through different combinations of detachable and attached types of formal and actual arguments of suppliers' features. Rule 2 states that lock passing only takes place if the corresponding formal argument is attached. From section 3.2 we know that an attached formal argument is locked by the routine. So, a routine that declares an attached formal argument will always lock that argument, either by waiting for the corresponding processor to become free (if its client does not hold the lock), or by enforcing lock passing from the client (if the client already holds the lock). Figure 8 illustrates these two cases. Since feature f in class X takes an attached argument, $x.f\ (y)$ will result in lock passing whereas $x.f\ (z)$ will be executed asynchronously without lock passing (just like in original SCOOP). This is because the client holds a lock on y but no lock on z.

```
-- in class C
z: separate Y
...
r (x: separate X; y: separate Y)
    do
        x.f (y)      -- Lock passing occurs because
                     -- Current has lock on y.

        x.f (z)      -- No lock passing because
                     -- Current has no lock on z.

        x.g (y)      -- No lock passing because
                     -- g takes detachable argument.

        x.g (z)      -- No lock passing because
                     -- g takes detachable argument.
    end

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

g (y: ?separate Y)
    do
        ...
    end
```

Fig. 8. Lock passing

If the called feature takes a detachable formal argument, as feature g in class X in Figure 8, no lock passing is performed. This logically follows from

the fact that such a feature does not lock the formal argument, as expressed by the locking rule in section 3.2. Since no lock is necessary, no lock passing takes place, independently of whether the client holds a lock on the corresponding actual argument (e.g. *y*) or not (e.g. *z*). Naturally, if a routine takes a detachable formal argument, it is possible to pass a detachable entity as actual argument (obviously, no lock passing takes place there because the client cannot hold a lock on a detachable entity). The opposite situation, i.e. passing a detachable actual argument to a routine that takes an attached formal argument, violates the conformance rules of detachable and attached – it is rejected by the compiler. Figure 9 recapitulates 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").

	actual locked by client	actual not locked
actual attached, formal attached	yes	no
actual attached, formal detachable	no	no
actual detachable, formal detachable	no	no

Fig. 9. Lock passing combinations

Coming back to the problem of backward-compatibility of our approach with the original SCOOP model, we can see that our new semantics corresponds to the original one in case where actual argument is not locked by the client. On the other hand, if the client holds a lock on actual argument, our semantics differs from SCOOP's. Nevertheless, it is possible to emulate the original semantics – at a cost of some additional code – through the use of detachable formal argument and an auxiliary feature that takes an attached formal argument, as illustrated in Figure 10. The call to $x.f\ (y)$ does not block, even though the client holds a lock on *y*. The call to *blocking_f* will later block the supplier but it does not influence the execution of our client's code. Therefore, we obtain the semantics of the original SCOOP model. Note the use of object test for a downcast from detachable to attached in feature *f*.

5 Related work

This paper builds on our previous work on locking policy for SCOOP described in a technical report [8]. The report discussed the use of detachable types in SCOOP but did not cover the problems related to inheritance and polymorphism, such as contract inheritance. The lock passing mechanism was only described shortly, without considering more complex scenarios that we discuss in this paper. The report also presented a basic mechanism for shared locking based on a refined notion of pure query and a new semantics for **only** clauses. The shared-locking mechanism proved unsound in the presence of polymorphism, therefore we do

```
-- in class C
z: separate Y
...
r (x: separate X; y: separate Y)
    do
        x.f (y)      -- No lock passing because
                     -- f takes detachable argument.

        x.f (z)      -- No lock passing because
                     -- f takes detachable argument.
    end

-- in class X
f (y: ?separate Y)
    local
        y': separate Y
    do
        if {y': separate Y} y then
            blocking_f (y')
        end
    end

blocking_f (y: separate Y)
    do
        ...
    end
```

Fig. 10. Emulating original SCOOP semantics

not consider it in this paper. We are currently working on a refinement of that mechanism that provides full support for inheritance and polymorphism.

Meyer [7] discusses detachable types, in particular their use for eliminating *catcalls*. He also describes the idea of using detachable types in the context of SCOOP which 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 prompted us to dig into the issue of contract redefinition 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. Nevertheless, 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. For example, for attached x and detachable y, expression $x.is\_empty$ **and** $y.is\_empty$ is understood as $x.is\_empty$ **and** $(\{y'\!:\ Y\}y$ **implies** $y'.is\_empty)$, hence $x.is\_empty$ if y is void and $x.is\_empty$ **and** $y.is\_empty$ otherwise. Besides being complicated, this solution is inconsistent with Design by Contract – as we demonstrated in section 3.3, it

may lead to postcondition weakening. Our refined rule for result and argument redefinition (Rule 1') may be combined with Meyer's solution to ensure consistency with DbC and to simplify his approach. This shows that our technique, initially developed to solve concurrency issues, proves very useful in a sequential context as well.

```
-- in class C
r (x: separate X; y: separate Y)
        -- We assume that y = x.my_y, so both Current and x will
        -- call the same separate object.
    do
        x.f             -- Body of f will request lock on y.

        -- Will lock passing happen here?
        y.f
        -- Or here?
        y.g
        -- Or here?
        y.h
        -- Or here?
    end

-- in class X
my_y: separate Y

f -- Perform some calls on my_y.
    do
        ...
        s (my_y)    -- Snatch lock from client.
        ...
    end

s (y: separate Y) do ... end
```

Fig. 11. Problems with transitive locking

Brooke et al. [2] propose a CSP semantics for SCOOP. The authors identify the problem of repetitive locking and propose to solve it by applying transitive locking by default. That is to say, if client object c holds locks on supplier objects x and y and x requests a lock on y, x will temporarily "snatch" that lock from the client object. An advantage of transitive locking is that it offers more potential parallelism than our solution (we apply synchronous semantics to calls that involve lock passing). Nevertheless, the programmer has no control over lock passing; transitive locking is always applied, even if there is no danger of deadlocking. Furthermore, it is possible that calls on y issued by c and x are interleaved: even though c temporarily loses its lock on y, it is impossible to

predict when it happens. If c executes several calls on y after the call on x (see Figure 11), lock passing may occur either before the call to $y.f$, before the call to $y.g$, before the call to $y.h$, or after the latter. In fact, lock passing may even not happen at all – if the execution of routine f by x is very slow then the client object may be able to schedule all its calls on y and terminate the body of r before x tries to snatch the lock. As a result, one cannot assume the order of execution of separate calls; hence, assertional reasoning about separate calls is not possible. This problem is particularly acute in the context of inheritance and polymorphism: even if the original version of routine f in class X does not perform any calls that would lead to lock passing, a redefined version may do so. Clients of the original class are completely unaware of that and do not expect any lock passing. Our solution avoids such problems: clients make explicit decisions about lock passing; a redefined version of a routine cannot require more locks than the original version. Furthermore, Brooke et al. only allow locks on separate objects to be passed to the supplier. As a result, the separate callback depicted in Figure 5 still leads to a deadlock. Another difference w.r.t. our solution is the fact that only one lock is passed at a time; nevertheless, additional locks can be demanded by subsequent calls. The CSP 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.

Rodriguez et al. [9] enrich JML with annotations for specifying atomicity and synchronisation constraints. Features can specify a list of locks that they acquire and release during their execution. A *locks* clause may appear in the specification of a feature after a precondition. By default, the locks clause has value *\nothing* for a non-synchronised feature. For features declared as synchronised, *locks* evaluates to *this* (or the class object if the considered feature is static). Another predicate, *lock_protected* (<*o*>), is added to the specification language. It allows to state that a given object (*o*) is protected by a (non-empty) set of locks, and that all these locks are held by current thread. Further, predicate *thread_local* (<*o*>) marks objects as thread-local, i.e. only reachable by current thread. Thread-local objects correspond to non-separate objects in SCOOP. Accesses to thread-local objects do not interfere with the activity of other threads, so they do not need to be synchronised. Although locking is specified at the feature level, it is more fine-grained than in our approach – locks are applied to single objects rather than whole processors. Also, lock passing is naturally supported. Nevertheless, the support for inheritance and polymorphism is lacking, e.g. it is possible to cheat on clients by performing more locking in a redefined version of a feature.

In Boyapati and Rinard's Parametrized Race-Free Java (PRFJ) [10], to gain an exclusive access to an object, a thread has to acquire the lock on the root of the ownership tree that contains the object. Every object has an owner: an object (possibly the object itself) or *thisThread*. If an object is owned by *thisThread* (directly or indirectly), it is local to the corresponding thread and it cannot

be accessed by other threads. Ownership is fixed, i.e. objects cannot change their owners over time. This ownership relation is very similar to the ownership relation between processors and objects in SCOOP, although the ownership structure in SCOOP is much simpler because objects cannot be owned by other objects and ownership is not transitive. A method may require callers to hold one or more locks before calling it – the locking requirements can be specified using the *requires* clause. Although PRFJ does not support Design by Contract, we may view *requires* annotations as part of routine contract. The support for inheritance and polymorphism is very limited – PRFJ does not offer the same flexibility w.r.t. routine redefinition as our approach. Also, unlike our approach, PRFJ does not support the combination of condition synchronisation and atomic locking of several objects.

6 Conclusions

We presented two simple refinements of the access control policy for SCOOP. We proposed a mechanism for specifying which arguments of a routine call should be locked. This mechanism, based on the novel concept of detachable types, allows for a precise specification of locking requirements, thus eliminating unnecessary locking that is often exhibited in SCOOP programs. We also introduced a lock-passing mechanism that allows clients to temporarily pass on their locks to separate suppliers. Both proposed mechanisms greatly improve the flexibility of the model and reduce the danger of deadlocks. They allow programmers to efficiently implement synchronisation scenarios that were difficult (or impossible) to implement in the original SCOOP model.

The *scoop2scoopli* preprocessor and the *SCOOPLI* library [1] support the lock-passing mechanism. We tested these tools in two iterations of a graduate course at ETH Zurich. A deadlock-detection scheme that supports lock-passing has also been devised and implemented [11] as part of SCOOPLI. We are currently working on the implementation of detachable types.

Our future research will be focused on the enhanced type system for SCOOP [12][13] and its applications to deadlock prevention. We think that the assertion language of Eiffel needs to be enhanced to allow for more expressive contracts for concurrency. In particular, we would like to enrich the specification of frame properties and use a refined notion of pure query to allow for safe interleaving of pure queries requested by different clients. We have already proposed the basic mechanism for shared locking [8] but more research is necessary to ensure its compatibility with the principles of Design by Contract.

7 Acknowledgements

Bertrand Meyer largely contributed to the development of the detachable/attached type mechanism and suggested its possible application in the context of

[1] Available for download at *http://se.ethz.ch/research/scoop.html*

SCOOP. Bernd Schoeller pointed out the problem of precursor calls in the presence of contravariance. We are grateful to the participants of the 2^{nd} SCOOP workshop for the discussions concerning the locking policy of SCOOP.

References

1. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice Hall (1997)
2. Brooke, P.J., Paige, R.F., Jacob, J.L.: A CSP model of Eiffel's SCOOP. Submitted for publication (2005)
3. Meyer, B.: Applying "Design by Contract". IEEE Computer **25** (1992) 40–51
4. Nienaltowski, P., Meyer, B.: Contracts for concurrency. In: First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE), York, United Kingdom (2006)
5. Caromel, D.: Towards a method of object-oriented concurrent programming. Communications of the ACM **36** (1993) 90–102
6. ECMA: Eiffel analysis, design, and programming language. ECMA Standard 367 (2005)
7. Meyer, B.: Attached types and their application to three open problems of object-oriented programming. In: European Conference on Object-Oriented Programming. (2005) 1–32
8. Nienaltowski, P.: Refined access control policy for SCOOP. Technical Report tr511, Computer Science Department, ETH Zurich (2006)
9. Rodriguez, E., Dwyer, M., Flanagan, C., Hatcliff, J., Leavens, G.T., Robby: Extending JML for modular specification and verification of multi-threaded programs. In: European Conference on Object-Oriented Programming (ECOOP). (2005) 551–576
10. Boyapati, C., Rinard, M.: A parametrized type system for race-free java programs. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (2001)
11. Moser, D.: Design and implementation of a run-time mechanism for deadlock detection in SCOOP. ETH semester project (2005) available at http://se.inf.ethz.ch/projects/daniel_moser.
12. Nienaltowski, P.: Efficient data race and deadlock prevention in concurrent object-oriented programs. In: OOPSLA'04 Companion. (2004) 56–57
13. Arslan, V., Eugster, P., Nienaltowski, P., Vaucouleur, S.: Scoop: concurrency made easy. In Meyer, B., Schiper, A., Kohlas, J., eds.: Dependable Systems: Software, Computing, Networks. Springer Verlag (2006)