Concurrency Annotations and Reusability

Klaus-Peter Löhr

Report B-91-13
November 1991

Abstract

Widespread acceptance of concurrent object-oriented programming in the field can only be expected if smooth integration with sequential programming is achieved. This means that a common language base has to be used, where the concurrent syntax differs as little as possible from the sequential one but is associated with a "natural" concurrent semantics that makes library support for concurrency superfluous. In addition, not only should sequential classes be reusable in a concurrent context, but concurrent classes should also be reusable in a sequential context. It is suggested that *concurrency annotations* be inserted into otherwise sequential code. They are ignored by a sequential compiler, but a compiler for the extended concurrent language will recognize them and generate the appropriate concurrent code. The concurrent version of the language supports active and concurrent objects and favours a declarative approach to synchronization and locking which solves typical concurrency problems in an easier and more readable way than previous approaches. Concurrency annotations are introduced using *Eiffel* as the sequential base.

Key words

Concurrent object-oriented programming, reusable concurrent code, concurrency annotations, Eiffel, CEiffel

Institut für Informatik
Fachbereich Mathematik
Freie Universität Berlin
Nestorstr. 8-9
D-1000 Berlin 31

lohr@inf.fu-berlin.de

1 Introduction

The recent surge of concurrent object-oriented languages indicates the lively interest of the research community in the subject, but is still far from being practically relevant. This is partly due to the immaturity of the subject; but another reason is that having a good language, although important, is not sufficient for practical software development: there is also a vital need for a stable programming environment comprising extensive libraries and powerful tools.

Now it is certainly too early for a generally accepted language of this kind to emerge, along with libraries and tools. But then, even with concurrent variants of well supported sequential languages like Smalltalk [Yokote/Tokoro 87] or C++ [Gehani/Roome 88], there remains the problem of reusability of classes across the boundary between sequentiality and concurrency: reusing sequential code in a concurrent system frequently requires a substantial amount of modification, involving special syntax and/or features from a "concurrency library"; and reusing concurrent code in a sequential context is not even considered.

A look at functional programming languages reveals that this need not be so. Slight changes in the semantics of a functional language, such as replacing eager evaluation with lazy evaluation, can suggest a concurrent execution model rather than a sequential one, in addition to enhancing the expressiveness of the language. In a similar vein, the ideal object-oriented language would come with a syntax that would allow either a sequential or a concurrent interpretation, depending on the compiler (or compilation switches) being used.

Our goal is approximation, if not attainment, of this ideal. In particular, it is important that the concurrent semantics blend well with inheritance, as a key to reuse. According to the terminology in [Papathomas/Nierstrasz 91], the approach reported here is *heterogeneous* and supports *concurrent objects* and *proxies*: i.e., objects may be active, may be threaded, may synchronize incoming requests and may support asynchronous service execution. Two languages known for similar properties are SINA [Tripathi/Aksit 88] [Aksit et al. 91] and ACT++ [Kafura/Lee 90]. The emphasis here, however, is less on concurrent language design and more on a common language framework accomodating both sequentiality and concurrency.

Our approach does not hinge on a particular language. Obviously, though, not all languages are equally well suited. We have chosen *Eiffel* [Meyer 88] as our experimentation vehicle, for reasons that will become evident below. Examples will be based on version 3 of the language [Meyer 91].

Eiffel has been used as the basis for concurrent programming before. A system called Eiffel// [Caromel 90] uses a slightly modified compiler and a library class `PROCESS`; concurrent objects are not allowed. Another system [Colin/Geib 91] relies completely on library classes; it is more flexible, but at the expense of cumbersome programming and poor reusability. A considerably modified version of Eiffel, called Distributed Eiffel, is described in [Gunaseelan/LeBlanc 91].

The system described here relies heavily on *annotations* to be inserted into otherwise sequential Eiffel text. These "concurrency annotations" have the form of Eiffel comments which are ignored by the (sequential) Eiffel compiler. They become "concurrently significant", however, if interpreted by a compiler supporting a concurrent semantics. In addition, the concurrent interpretation of a given program text occasionally is different from the sequential interpretation even if no annotation is directly involved. The annotated version of the language is called *CEiffel*.

Section 2 motivates and describes the concurrency annotations, their interdependence and their interplay with inheritance. Delayed execution of operations on objects and its relation to exceptions is the subject of section 3. Contention on access to objects raises scheduling questions; how to implement, inherit and redefine non-standard scheduling strategies will be discussed in section 4. Our work on concurrent object-oriented programming is part of a larger effort to support the distributed execution of object-oriented programs in a heterogeneous network (project HERON). This context and the status of the project will be described in section 5. Comparison with related work can be found throughout the paper. For an overview on current trends in concurrent object-oriented programming see [Papathomas/Nierstrasz 91] and [Agha et al 91].

2 Concurrency annotations

Before turning to Eiffel we introduce an informal object model together with some basic terminology, trying to capture most of the commonly used notions for object-oriented concurrency while avoiding a bias towards any specific language.

2.1 Operations, activities and active objects

Each class has a set of *operations*: they define possible state transitions of any given object of that class from one abstract state to another; they also provide for information flow between the object and its environment. How this is accomplished depends on the representation (the concrete state) and is described by the code of the class. For the present discussion we do not distinguish between "class" and "type". Remember, however, that the notions of inheritance and subtyping are not identical [Cook et al. 90] [America/van der Linden 90] [Meyer 91]. It should also be kept in mind that it is crucial for the development process not only to distinguish between a class and its signature, but also to clearly identify its specification.

An activation of an operation is called an *activity*. At any given time, an object is either *idle*, i.e., with no current activity, or *busy*, i.e., there is one activity or multiple concurrent activities. Note that concurrent activities of an object may have a combined effect that cannot be achieved by any serial execution of those activities. If a class imposes no restrictions on multiple activities for its objects, it is called a *concurrent class*; an object of that class is called a *concurrent object*. If multiple activities are not allowed the class is called *atomic* (and so are the objects).

An activity starts when a corresponding *request* has arrived and is *accepted* by the object. There are two ways of how requests are generated:

1. A so-called *autonomous operation* issues a request for itself as soon as the object has been created and initialized. When the activity terminates the request is re-issued. An autonomous operation has an empty signature. A class or an object that has autonomous operations is called autonomous as well.

2. Requests for non-autonomous operations are issued by other objects through operation *invocation*. The originator of the invocation is called the *client* of that invocation, the invoked object is called the *server*.

A request that has been issued but not been accepted yet is said to be *pending*. When an activity terminates it generates a *reply* (if the operation has no result, the reply carries no value)[1].

Invocation raises the issue of how the client activity and the server activity are related. Sequential semantics postulates nested execution: after having generated the request the client activity waits for the reply. But in a concurrent environment the client may also be

[1] We exclude the possibility of sending a reply before termination, for reasons to be explained later.

allowed to proceed after the acceptance - or even immediately after the invocation - and to synchronize with the reply later, if necessary. This is commonly known as client/server *asynchrony*. If asynchrony is declared a property of the operation (as opposed to being caused by the client), the operation is called *asynchronous,* as is the corresponding class and its objects. Note that a class/object can be both autonomous and asynchronous.

An object that is autonomous or asynchronous is called an *active object*; the others are called *passive*. Active objects are sources of a varying number of concurrently executing activities in a running system. Note that we treat passiveness vs. activeness on the one hand and atomicity vs. non-atomicity on the other hand as independent issues (in contrast with other approaches known from the literature). We avoid notions like "process" or "thread" and defer the question of how to implement active objects until later.

There are ways to simulate autonomy by asynchrony, but it is not natural to do so, especially when inheritance is involved. Autonomy makes it possible to model autonomous entities which need not be triggered from the outside in order to become active. Asynchrony is mainly used to achieve speedup, which of course depends on the underlying system architecture. Autonomy causes *horizontal concurrency* whereas asynchrony causes *vertical concurrency* (within a functional hierarchy).

2.2 Asynchrony

In Eiffel an operation is represented by an exported feature, i.e., a routine or an attribute. Functions and attributes deliver results, procedures do not. The two classes involved in a client/server relationship between objects are called client class and supplier class. As Eiffel is sequential, there are no autonomous or asynchronous operations, and consequently no active objects.

An obvious way to interpret Eiffel code as concurrent code is to consider *all* exported routines asynchronous and to use *lazy synchronization*: upon invocation of a function a result is returned immediately, but this result is just a proxy for the expected reply; later on, the first operation on that proxy implies a synchronization with the delivery of the real result upon termination of the corresponding activity. Similar techniques are known from other languages [Papathomas/Nierstrasz 91]; Eiffel// uses the term *wait-by-necessity* [Caromel 90].

Now while it is certainly possible to write meaningful concurrent programs in such a variant of the language, serious objections remain. First, due to the rather fine-grained concurrency caused by a plentitude of small routines, *efficiency* will most likely be so poor as to defeat the very purpose of introducing concurrency in the first place. Secondly, and at least as important, writing programs that behave correctly under the concurrent interpretation will not be easy. The programmer has to be very careful to avoid unplanned interference between the concurrent activities. Such interference looms everywhere, even if the system does not contain concurrent objects. The innocent-looking code

```
r := server.computation1(y); -- asynchronous --
computation2(z);
r.p;                         -- synchronization --
r.q;
```

may have weird effects if `server` or `y` are involved in `computation2`. The most important point, however, is that the concurrent semantics of this code may be so different from its sequential semantics that we miss our goal - reuse across the sequential/concurrent boundary.

Rejecting implicit asynchrony for CEiffel, we attach an *asynchrony annotation* to a routine that is to be executed asynchronously under concurrent interpretation. The annotation is written as a comment `--v--` which is ignored under sequential interpretation[2]. The `v` may be read as a downward arrow or as "vertical concurrency". The following example demonstrates the use of the annotation:

```
computation1(y: T1): T2 is --v--
do ..... end; -- computation1 --
```

Both the class and the objects are said to be asynchronous in this case. After an invocation of *computation1* the client proceeds immediately, even before the request is accepted[3]. Lazy synchronization is used in claiming the result. - The annotation is ignored in local calls of the routine.

Note that asynchrony is not just an implementation property of an operation. The client must know about asynchrony in order to avoid undesirable interference with the asynchronous activity. There are other (rare) cases where client and server have to *cooperate* to achieve a common goal which means that "interference" is mandatory rather than unwanted. This is why asynchrony is introduced as a property of an operation rather than the effect of a *fork* operation. In any case, asynchrony must be considered part of the *specification* of a class.

---

[2] We pretend that an Eiffel comment which starts with `--` also ends with `--`. This is *not* so; it ends with the line end. But observing this would force us to use a poor layout in the examples below.

[3] As opposed to a "synchronous send" or a rendezvous-like interaction between client and server this requires buffering of requests but is better suited for distributed implementation (cf. section 5).

2.3 Autonomy

There is no satisfying way of automatically identifying autonomous operations under a concurrent semantics. Viewing non-exported routines with an empty signature as autonomous is about the closest we can get. But this would sometimes force us to introduce dummy signatures and thus would also hamper reuse of existing sequential classes.

Our choice for CEiffel is again using an annotation. The *autonomy annotation* is written `-->--`. The annotation can only be attached to a routine with an empty signature, as in

```
action is -->--
do ..... end; -- action --
```

Autonomous routines are usually not exported; occasionally, though, export may be useful, e.g., for testing purposes in a sequential environment. The concurrent semantics of an autonomous routine is exactly as explained in 2.1, although with a special proviso: the invocation of an exported autonomous routine has no effect.

A class may feature several autonomous routines. Also, there may be both asynchronous routines and autonomous routines. Each routine, however, is either synchronous or asynchronous or autonomous. Let us consider the example of a class `Moving` which captures properties of moving bodies in two-dimensional space; it might be used in a simple animation system. The velocity of a `Moving` object can be "remotely controlled". The given code ignores the actual display programming and any alignment with real time.

```
class Moving
creation create
feature -- interface --
        position: Vector;

        setVelocity(v: Vector) is
        do velocity.set(v.x,v.y) end;
```

```
feature {} -- hidden --
        velocity: Vector;

        step is -->--
        do position.set(position.x + velocity.x*stepTime,
                        position.y + velocity.y*stepTime) end;

        create(startingPoint: Vector; timeUnit: Real) is
        require timeUnit > 0
        do position := startingPoint;
           stepTime := timeUnit end;
end -- Moving --
```

Note that `Moving` must be atomic (it *is* atomic indeed, as explained in 2.4 below). If it were concurrent, overlapping `setVelocity` activities could have nasty effects: concurrently setting the velocity to `(0,1)` and `(1,0)` might produce the velocity `(0,0)` (depending on the implementation of `Vector`). However, we might consider declaring `setVelocity` compatible with `step` and `position`. One can readily see that execution of `setVelocity` while `step` is in progress causes only a minor irritation in the movement which we might find acceptable. Concurrently reading `position` and changing it by `step` produces an equally acceptable result. - The class is of course fit for sequential interpretation, except that it would not be of much use without exporting `step`.

Asynchronous and autonomous operations offer several advantages over more traditional concepts such as a "body" describing the lifelong behaviour of an active object (see, e.g., POOL [America 87] [America 89], ABCL/1 [Yonezawa et al. 87] or Eiffel//). First, as a body constitutes a permanent thread of control, concurrent activities within an object are either excluded or have to be created explicitly by the body (by a mechanism similar to the `detach` in SINA). Secondly, every request must be explicitly accepted by the body (except if the body is omitted - but then the object cannot be autonomous). This is not only cumbersome for the programmer, it is incompatible with multiple inheritance, because the body has to be redefined; even with simple inheritance, redefinition is almost always required. A further disadvantage is the fact that functional hierarchies of such objects are prone to the same pitfalls as known from nested monitors.

Autonomous operations do not suffer from these problems. Note that the semantics of an autonomous operation is *not* identical to that of a body containing a corresponding loop. It is also beneficial that asynchrony is not tied to the presence of a body because asynchrony and atomicity are independent issues.

2.4 Concurrent classes and controlled objects

Under sequential interpretation all classes are de facto atomic. This property should be maintained under concurrent interpretation *unless* concurrency is explicitly allowed. Declaring two operations *compatible* allows them to be executed concurrently. Compatibility is declared by means of *compatibility annotations* `--‖...--` attached to the relevant routines. E.g., we might find the following declarations in a class `Queue` written in CEiffel:

```
enqueue(x: T) is   --‖ dequeue, length --
do ..... end;
dequeue: T    is   --‖ enqueue, length --
do ..... end;
length: Integer is --‖--
do ..... end;
```

These annotations express that the class has been *implemented* in such a way that a certain overlapping of activities (e.g., of an `enqueue` activity and a `dequeue` activity) can safely be allowed, i.e., does not violate the specification of the class. Compatibility is always a symmetric relation; redundant compatibility information can be omitted in the annotations. If no name is given in an annotation, the routine is compatible with itself and all other routines annotated in this way. An operation implemented as an attribute is "implicitly annotated" with `--‖--`. The explicit `--‖--` is typically used for read-only operations which do not change the state of the object.

Compatibility annotations are of course independent of asynchrony annotations. E.g., we might add `--v--` to the declaration of `enqueue` (although it is probably not worth the effort, given the simplicity of the operation).

A general compatibility annotation `--‖--` can be attached to the class head; this is shorthand for expressing that everything is compatible, i.e., the class is fully concurrent. If no compatibility annotation and no exported attributes are present, the class is atomic. A class that is neither atomic nor fully concurrent is said to be *semi-concurrent*.

The compatibilities of an atomic or semi-concurrent class are relevant in those cases only where a fully concurrent version of the class could lead to unwanted overlapping of activities. The programmer has to decide whether or not an object should be subject to concurrency control according to the compatibilities stated in the class. The *control annotation*, written `--!--`, can be attached to a declaration, as in

```
q: Queue[Message] --!-- ;
```

It affects the creation of `q`, declaring that the object is to be controlled[4]. For our semi-concurrent `Queue`, appropriate locking mechanisms are automatically built into the object. If `Queue` were atomic, `q` would refer to an atomic object (which is akin to a monitor or a sequential process). The control annotation is a type qualifier: type conformance is violated if one type is annotated and the other is not.

Locking in CEiffel is a generalization of read/write locking as employed in Distributed Eiffel [Gunaseelan/LeBlanc 91]. When an atomic or semi-concurrent object is busy, requests that are incompatible with existing activities remain pending as defined in 2.1. As soon as the termination of an activity causes requests to become eligible for acceptance a standard scheduling strategy applies: pending requests are accepted in arrival order (FCFS). Non-standard scheduling can overcome this; for details see section 4.

Controlling is rare. Most objects are not controlled, in particular the vast amount of sequentially used objects in a program. Also note that there is not even a need to control every object that is shared among concurrent activities. E.g., a controlled semi-concurrent object may have component objects that are shared, yet uncontrolled. The usage pattern of a shared object ultimately determines whether control is necessary or not. E.g., if a `Queue` object serves as a buffer between only one producer and one consumer, `--!--` can be omitted.

Repeated invocations of an asynchronous object may lead to multiple activities. For this reason, asynchronous objects are *always* controlled, even if not annotated. Analogously, autonomous objects are always controlled. Thus, active objects that are not fully concurrent are *always* furnished with built-in locking mechanisms.

Controlled objects may not be read or written (i.e., compared, copied, assigned to), due to the possible interference with their exported operations. It is, of course, allowed to read/write references to such objects. These read/write operations are indivisible, as are all read/write operations on objects of the basic types (`Boolean` etc.). The library class `Array[T]` is concurrent; `item/put` operations behave like reads/writes on uncontrolled `T` objects.

If a concurrent class inherits from other classes, its routines are compatible with the inherited routines, in addition to being compatible among themselves. However, no new compatibilities among the inherited routines are introduced. With multiple inheritance, any routine of a parent is compatible with any routine of another parent.

---

[4] Note that if `Queue` is an "expanded type" the creation is implicit and `q` denotes the object. If `Queue` is not expanded, the object must be created explicitly and `q` denotes a reference to the object.

2.5 Asynchrony, autonomy and inheritance

Inheritance works for asynchronous and autonomous routines as for any other feature. Let us look at a simple example. Imagine a class `Beeping` that captures the property "repeatedly generating a beep sound" where the beeping can be turned off and on. A simple version is

```
class Beeping
creation create
feature -- interface --
        on(b: Boolean) is
        do beepon := b end;

feature {} -- hidden --
        beepon: Boolean;
        sound:  Speaker;

        beep is -->--
        do if beepon then sound.beep end end;

        create(s: Speaker) is
        require s /= Void
        do sound := s end;
end -- Beeping --
```

Now if we want to capture the properties of objects both moving and beeping we can use multiple inheritance. The resulting class has several synchronous routines and two autonomous routines. By introducing additional attributes we can design, say, a class `Mouse` that captures the properties of a - still rather abstract! - rodent:

```
class Mouse
inherit Moving rename create as mcreate end;
        Beeping rename create as bcreate end
creation create
feature .....
end -- Mouse --
```

Feature adaptation (like renaming, redefinition, changing the export status etc.) applies to asynchronous and autonomous routines as to any other feature. Redefinition and effecting (of a deferred function), collectively known as *redeclaration*, deserve special mentioning. Changing the concurrency property in a redeclaration is allowed, although rare. Typical cases are:

- A deferred routine carries no annotation. The corresponding effective routine is marked `--v--` .
- An effective routine is marked `--v--`. A major reorganization of some features allows the former routine to be redeclared as an attribute (which never carries a `--v--`).
- A descendant reduces the amount of autonomy inherited from an ancestor by redefining an autonomous routine `auto` as `auto` **`is do end`**.

Attaching a concurrency annotation to a deferred routine is not prohibited, although it is of mere declamatory value. E.g., the asynchrony annotation would specify that the routine has to cooperate with the client, as mentioned in 2.2.

Note that an inherited routine can have different concurrency properties in different heirs. E.g., let `C` be a parent of `A` and `B`, with a deferred routine `op`. `A` might declare `op` asynchronous while `B` might not. Due to the polymorphism we cannot tell (and do not care) whether the call `c.op` (with `c` of type `C`) will generate a synchronous or an asynchronous activity.

3 Delayed requests

3.1 Preconditions and delays

An operation with a non-empty *precondition* represents a partial function with a domain characterized by the precondition. A precondition can state consistency requirements for parameters or may restrict the states in which the operation can meaningfully be executed; it may also involve both parameters and state.

A violated precondition should raise an exception in a sequential environment. For a shared object in a concurrent environment, a precondition involving the state should sometimes act as a *guard* rather than a source of exceptions: an incoming request should be *delayed* if, and as long as, the precondition is violated *and* can be satisfied by changing the state. A delayed request is pending (as defined in 2.1) and cannot be accepted until the precondition is satisfied. The rationale for delaying is that if state is involved the precondition might become satisfied through the effects of other activities. Of course, there is no guarantee for this, and the issue has to be studied in more detail. We will come back to this in a moment.

A classical example is the finite queue. Overflow or underflow of a sequential queue should raise an exception. But under concurrent interpretation, a shared queue should act like a message buffer where delays prevent overflows and underflows. A generic Eiffel class `Queue` might look like this:

```
class Queue[T]
        .....
feature
        enqueue(x: T) is require length < maxlength
        do ..... end;

        dequeue: T    is require length > 0
        do ..... end;

        maxlength: Integer is ...;

        length: Integer is
        do ..... end;
        .....
end -- Queue --
```

This class has both a sequential and a concurrent interpretation, the only difference being exceptions vs. delays. Note that if we wanted to declare `enqueue` (or even `dequeue`) as asynchronous we could do so, but, as mentioned in 2.4, it is probably not of much use.

Since the class is atomic, operations on a controlled `Queue` object behave like conditional (better "delayed") critical regions and the object behaves like a delayed monitor with standard scheduling: acceptable requests are served in arrival order. In this case, the favourable moment for reevaluating the preconditions of delayed requests is when an activity terminates. We also use this technique for non-atomic objects, for efficiency reasons. Remember that the cost of reevaluation can be kept low by appropriate compilation techniques [Schmid 76].

It should be kept in mind that in non-atomic objects all the usual interference problems are compounded by the fact that precondition evaluation may overlap with other activities. This may even cause different values to be observed for an attribute that is referred to several times in a precondition. But also remember that sensible non-atomic implementations of objects do exist and that highly parallel data structures are an active research subject [Herlihy/Wing 90] [Herlihy 90]. In fact, the above `Queue` has a straightforward semi-concurrent implementation as suggested by the compatibility annotations in the example in 2.4.

Preconditions can favourably be used in conjunction with autonomous operations. Actually, they are the primary means for controlling autonomous operations. E.g., the autonomous operation `beep` in the class `Beeping` is much better expressed as

```
class beep is -->--
require beepon
do sound.beep end   .
```

In some concurrent languages an activity can send a reply *before* it terminates; this allows the client to continue while some "postprocessing" is performed by the server. Ada is an early example for this, POOL is another one. We cannot support this because it is incompatible with the Eiffel style of returning a result (by assignment to the predefined entity `Result`). But we can easily simulate it by splitting the operation into a replying operation and a delayed autonomous operation. Admittedly, this amounts to misusing horizontal concurrency for vertical concurrency, and it negatively affects reusability: the sequential interpretation is unusable without a redefinition of the replying operation.

A typical example is a `Repository` object resembling a cloak-room: items can be deposited in exchange for a "ticket"; the item is actually "stowed" after handing out the ticket. When picking up the item later, the ticket has to be presented (and is invalidated).

```
class Repository[C]
        .....
feature deposit(item: C): T is
        require spaceAvailable and place = Void
        do place := item;
           Result := getTicket end;

        stow is -->--
        require place /= Void
        do ..... -- stow contents of place --
           place := Void end;

        pickup(ticket: T): C is
        require ticket.valid
        do ... -- hand out item and invalidate ticket -- end;

        .....
end -- Repository --
```

For a shared `Repository` object it is obvious that the preconditions of `deposit` and `stow`, referring to the object's state, must cause delays if violated. It is equally obvious that violation of the precondition of `pickup` must raise an exception.

3.2 Delays vs. exceptions and the delay annotation

A problem arises with a private `Repository` object: violation of the precondition of `deposit` should raise an *exception* if not `spaceAvailable` or else should cause a *delay* if not `place=Void`.

A similar problem occurs with a slightly different version of `Repository` where not even autonomy is involved:

```
deposit(item: C): T is
require spaceAvailable
do ..... end;

pickup(ticket: T): C is
require valid(ticket)
do ..... end;
```

Tickets are not reused; a ticket is invalidated by the very act of picking up (i.e., removing) the corresponding item. The precondition of `pickup` is state-dependent: the routine `valid` checks for the presence of an item associated with the given ticket. If this check fails, an exception has to be raised, regardless of whether the object is private or shared. So with a shared object `deposit` must produce delays while `pickup` must produce exceptions - although both preconditions refer to the state.

The examples demonstrate that the search for a completely automatic decision for delays vs. exceptions is futile. This motivates the introduction of a *delay annotation*, written `--@--`, which can be inserted into a precondition, between two assertion clauses[5]. It divides the precondition into two parts, the *exception part* and the *delay part*. The exception part of an autonomous routine must be empty. An object invocation that violates the precondition causes an exception if an assertion clause in the exception part is violated; otherwise, a delay occurs.

As a consequence, several preconditions in the above examples have to be annotated using `--@--` . Specifically, a variant of `enqueue` might read

[5] Remember that the keyword **`require`** is followed by a sequence of assertion clauses separated by semicolons which represent semi-strict "and then" operators.

```
enqueue(x: T) is
require x /= Void;
  --@-- length < maxlength
do ..... end;
```

The precondition of the first version of `deposit` may be written either

```
require
  --@-- spaceAvailable and place = Void
```

or

```
require spaceAvailable;
  --@-- place = Void
```

depending on the desired semantics.

3.3 Redeclared preconditions

Redeclaration of a routine may involve weakening the precondition. If an inherited routine with precondition

```
require A1;...;An
```

is redeclared with

```
require else B1;...;Bm
```

the effective assertion for the routine is

```
B1;...;Bm or else A1;...;An .
```

If this assertion turns out to be violated, the request is delayed if at least one of the disjuncts satisfies the criterion for delaying given above.

As an example, consider a class that manages printers of different types. There is a fast "standard" printer and a slow "special" printer that has special capabilities (say, colour) but includes the capabilities of the normal printer. An operation

```
get(needspecial: Boolean; size: Integer): Printer is .....
```

asks for a printer which is chosen on the basis of availability, capabilities and the size of the printing job. The precondition is

```
require size > 0;
  --@-- specialidle or standardidle;
        needspecial implies specialidle    .
```

If we want to accomodate a third printer, say a slow standard printer, we use inheritance and redefine the `get` routine. The precondition is weakened by

```
require else size > 0 and size < 5000;
     --@-- not needspecial and thirdidle   .
```

Only `size<=0` raises an exception, both with the original and with the redeclared `get`.

4 Scheduling

The sharing of atomic and semi-concurrent objects raises the question of how the acceptance of concurrent requests is to be scheduled. The default scheduling strategy is FCFS: acceptable (i.e., non-delayed) requests are accepted in the order they were issued. This strategy prefers a delayed request, as soon as it becomes acceptable, over a new acceptable request. It cannot, of course, prevent indefinite delays.

Concurrent objects, although not subject to scheduling, can exhibit a "scheduling anomaly" when an activity causes the precondition of a delayed request to become satisfied: since the precondition is only reevaluated when the activity terminates, a new request with the same precondition may be accepted immediately, thus overtaking the delayed one and maybe even causing it to be delayed further. The danger of starvation lurking here is one of the pitfalls of concurrent objects the programmer has to be aware of.

If non-FCFS scheduling is required it must be programmed explicitly. This task can be alleviated considerably by special language support which by its very nature exceeds the realm of a sequential programming language. Annotations cannot do the job any more. We suggest an approach that allows to refer to pending requests either in preconditions or in special scheduling routines.

4.1 The request list

We have to look at the semantics of shared and active objects in more detail. Associated with every such object is a *request list* which at any time contains the pending requests for that object. For each class, the predefined identifier `Request` denotes the type of the requests associated with that class. Data of a `Request` type cannot be passed between objects. A `Request` type is akin to a variant record type, with operations corresponding to the variants, signatures to the record fields and actual parameters to the actual record components.

In addition, every request has a component `Rank` of type `Integer`. `Rank` is the virtual arrival time of the request relative to the object's creation time: 1 for the first request, 2 for the second, etc. `Rank` can be used like a formal parameter.

When a request arrives at an object and is not immediately rejected (exception) it enters the request list. If the compatibilities allow a corresponding activity to be started, the activity is tentatively started, evaluating the precondition; if the precondition is satisfied, the request is removed from the request list and the activity continues; if not, the activity is aborted and the request remains in the request list (delay). When an activity terminates the pending requests are scanned in arrival order, and each request is treated as described for an arriving request.

Explicit modification of the request list is not possible, but the concurrent version of the language has several constructs for iterating over the request list, viz. a statement and three expressions:

```
for all   Feature_name do   Compound end
    all   Feature_name sat  Boolean_expression
    some  Feature_name sat  Boolean_expression
    that  Feature_name sat  Boolean_expression
```

The **for all** iterator executes the given Compound for all requests for an operation with the given Feature_name. The **all** iterator is the universal quantifier ranging over all requests with Feature_name; **some** is the existential quantifier. **that** refers to the least recent request satifying the given Boolean_expression. **sat** means "satisfy(ing)". Within the Compound or Boolean_expression, the Feature_name denotes the iteration variable (!) which refers to the different requests; argument names and dot notation allow to refer to the arguments of a request. E.g., if a class has an operation `get(n: Integer)` the following expression is legal:

```
all get sat get.n > 1     .
```

4.2 Scheduling routines

The concurrent version of the language knows about a predefined routine identifier `Scheduler`, and a class may contain a routine

```
        Scheduler: Request is do  ..... end    .
```

If the request list of an object of that class is non-empty, an activity of the object is automatically extended with an execution of `Scheduler` (so the `Scheduler` is somewhat similar to an autonomous routine). The scanning of the pending requests is then started with the request produced by the `Scheduler` (if not `Void`) rather than with the least recent request.

A typical example for non-FCFS scheduling is "smallest-request-next" for a collection of identical resources[6]:

```
  class SRNresources
  feature get(n: Integer) is
          require n>0;
            --@-- n<=available
          do ..... end;

          put(n: Integer) is do ..... end;

  feature {}
          Scheduler: Request is
          local min: Integer
          do min := MaxInt;
             for all get do
                 if get.n<min then
                    min := get.n;
                    Result := get end end end;
          .....
  end -- SRNresources --
```

Note that adding a scheduling routine does not modify the conditions for delaying requests. It just makes the class text explicit about the order that acceptable requests are accepted. Typically, inheritance will be used for adding a specific scheduling routine to a class. Imagine a generic class `Resources[T]` which might be used for managing, say,

[6] This happens to be an example that is easier solved with a **by** clause à la SR [Andrews et al. 88]. However, **by** is suitable for a limited class of problems only.

storage units in a sequential program. The same class can be used in a concurrent program, with no changes whatsoever, using standard scheduling. If some non-standard scheduling is required, say SRN, a class `SRNresources[T]` would be designed as an heir to `Resources[T]`:

```
class SRNresources[T]
inherit Resources[T]
feature {}
        Scheduler: Request is ..... end;
end -- SRNresources --
```

More complex inheritance hierarchies may necessitate redefinition and/or renaming of `Scheduler` routines, but the operations and their preconditions are usually not affected.

4.3 Scheduling through preconditions

There are cases where a specific scheduling strategy cannot be expressed by a `Scheduler` routine. Consider a class `Resources` (as above) with standard scheduling. As the FCFS strategy is applied to the *acceptable* requests, large requests may suffer indefinite delays, as mentioned before. If we insist on strict FCFS handling for the `get` requests we have to delay such a request if delayed `get` requests are already present, regardless of `n<=available`. We can have the precondition reflect this:

```
get(n: Integer) is
require n>0;
  --@-- n<=available;
        all get sat get.Rank>=Rank
do ..... end
```

One may argue that preconditions involving the state of the request list are more elegant than a `Scheduler` routine. E.g., `SRNresources` is readily implemented like this:

```
class SRNresources
feature get(n: Integer) is
        require n>0;
          --@-- n<=available;
                all get sat get.n>=n
        do ..... end;
        .....
end -- SRNresources --
```

This technique, however, does not blend well with inheritance. Modifying the scheduling strategy would require redefinition of `get`, but this is only possible with *weakening* the precondition. Besides, the above solution is less efficient than the `Scheduler` solution in 4.2. So while it is certainly possible to implement SRN as shown above (and even necessary to implement the FCFS example in this way), making a precondition refer to the request list is not generally recommended.

Precondition scheduling does work fine for the *Readers/Writers* problem which is an instructive example of how requests for non-atomic objects can be scheduled:

```
class ReadersWriters[T] -- for expanded types T --
feature read: T;        -- compatible with itself; see 2.4--

        write(t: T) is  -- incompatible with read, write --
        do read := t end;
end -- ReadersWriters --
```

The standard scheduling strategy described in 4.1, although basically FCFS, does not prevent readers from sneaking past writers when reading is in progress, effectively giving preference to the readers. Writer priority can be achieved by implementing `read` as a routine with a precondition:

```
class RW[T]
feature read: T is --‖--
        require
          --@-- not some write sat true
        do Result := data end;

        write(t: T) is
        do data := t end;

feature {}
        data: T;
end -- RW --
```

Strict FCFS scheduling (with multiple reading allowed) is achieved by

```
read: T is --‖--
require
  --@-- not some write sat write.Rank < Rank
do Result := data end;
```

Note that this precondition is weaker than that above and stronger than the empty precondition preferring the readers. We may conclude that writer priority is the right choice for a parent class `RW` that is to be reusable with modified scheduling strategies. We can even reconstruct the first version, changing the function `read` into an attribute by redeclaration:

```
class ReadersWriters[T]
inherit RW redefine read, write end
feature read: T;

        write(t: T) is
        do read := t end;
end -- ReadersWriters --
```

Of course, this is a somewhat sterile exercise, especially since `data` is not used, yet still present behind the scenes; but it is nice to see how the redefinition of `read` fits together with the weakened scheduling requirement.

4.4 Comparison with other approaches

Guide [Decouchant et al. 91] allows a class to be extended by a *control clause* which contains *activation conditions* for the operations. An activation condition acts like a guard. In addition to the operation's arguments and the state of the object, it may refer to *synchronization counters* associated with each operation, such as number of invocations, of acceptances, etc. There is no way to refer to the pending requests (apart from counting). Activation conditions can be redefined in subclasses.

The Guide approach mixes up two conceptually different issues: delays, which is a specification issue, and compatibility, which is an implementation issue. More importantly, delaying is mixed up with scheduling. If activation conditions have to be redefined in a subclass, considerable scrutiny is required to find out which parts pertain to delays, compatibilities and scheduling, resp. In addition, as a request cannot refer to the arguments of other pending requests, there is no easy way to implement non-trivial resource scheduling strategies.

PCM (for "priority-controlled modules") [Bahsoun et al. 90] uses two separate classes for the implementation part and the synchronization part of objects; the latter class is called a *synchronizer*. Inheritance is used to build synchronized classes. This approach works satisfactorily only for the simplest examples where the synchronization strategy is independent of the object's state and of the arguments of the operations. With Guide it shares the drawback that delays, compatibilities and the scheduling proper are mixed up. It has the additional disadvantage that a synchronizer is imperative (whereas Guide's control clause is declarative[7]).

PO (for "parallel objects") [Corradi/Leonardi 91] is a language in the Smalltalk tradition. A class can be extended by a *scheduling part*; it describes the activity of a scheduler which gains control when an activity terminates. As in Guide and PCM, everything pertaining to synchronization is indiscriminately handled by the scheduler. Since programming the scheduler is cumbersome, declarative "constraints" can be used for abbreviation purposes. As opposed to Guide and PCM, though, the scheduler can inquire about arguments of pending requests. The resulting flexibility allows, e.g., to insert two records into a table object concurrently if their keys are different. A subclass inherits the scheduling part of its superclass and may add further constraints.

5 Context and perspective

5.1 Project HERON

Smooth integration of sequential and concurrent object-oriented programming is of particular importance if distributed execution of programs is to be supported in a distribution-transparent manner. Basically, we take the view that distribution and concurrency are independent issues as far as the application programmer is concerned. This attitude is rooted in the remote procedure call paradigm (RPC) which allows transparent distribution of sequential programs. But then a slightly different view of RPC, associated with the term "remote invocation", is that of an inter-process communication facility. This view is tied to the notion of a server process and, if embodied in the programming language rather than confined to system-level processes, leads to concurrent application programs. Concurrency and distribution combined allow us to write parallel programs that exploit both shared-memory and networking parallelism.

Project HERON is the effort to develop a platform for the distributed execution of object-oriented programs in heterogeneous networks. It is a language-based approach to what is called Open Distributed Processing (ODP) by the ISO and the ECMA [ISO 90] and covers mainly the *computation viewpoint* and the *engineering viewpoint* of ODP.

HERON’s basic tenet is that the concurrency structure of an application system is not necessarily related to its distribution structure. The way different parts of the system are

[7] One might argue that the declarative style is less flexible. Actually, both styles should be supported - which is exactly what we are doing (see also [Caromel 91]).

distributed among different address spaces, and where in the network these address spaces are instantiated as system-level processes (possibly threaded) is not determined by programming but by an independent configuration procedure. HERON uses Eiffel and CEiffel both as the reference languages for application programming and as the implementation languages for the run-time support. The project relies on experience gained from DAPHNE, a module-based system for distributed execution of sequential Modula programs [Löhr et al. 88].

5.2 Implementation issues

A CEiffel program can be executed in a threaded address space. But only the most naïve implementation would come up with a one-to-one correspondence between activities and threads. Reusing threads from a common pool is an obvious optimization. But in some cases the compiler will be able to recognize that several activities can share a thread:

1. Non-remote concurrent passive objects: A server activity shares the client's thread, and inter-object communication is implemented as procedure call.

2. Atomic active objects: One thread is used for all activities of an object, and inter-object communication is implemented as message passing, possibly across machine boundaries.

HERON will support both single-address-space execution and distribution of programs among several threaded address spaces which may reside on different machines. Any remote invocation, i.e., an invocation across an address space boundary, will involve different threads. As opposed to Distributed Eiffel, the syntax and the semantics of CEiffel are not concerned with distribution issues. Regarding class texts and object creation and invocation, there is no difference between local and remote objects. A configuration tool takes care of distribution issues like stub generation and construction/placement of load images on different nodes of the heterogeneous network.

We have implemented a threading library for Eiffel which is based on coroutines and asynchronous Unix (SunOS) system calls. In order to accomodate heterogeneity, we have striven for a portable design, isolating a front-end from a system-specific back-end; the latter can take advantage from operating systems offering a true threading facility to user programs.

A prototype version of a concurrent Eiffel system is being implemented as a precompiler which generates threaded Eiffel code. Concurrently, run-time support and a stub generator are being developed for distributed execution.

5.3 Conclusion

The usage of concurrency annotations enables us to write classes both representing correct sequential Eiffel code and allowing for a concurrent interpretation. In most cases, a class can be used both in a sequential and in a concurrent context, and inheritance causes no surprises in concurrent programs. The annotations are:

`--v--` and `-->--` : asynchrony and autonomy
`--||...--` and `--!--`: compatibility and controlling
`--@--` : delay on assertion violation

We identified non-standard scheduling as an issue inherently tied to concurrency, justifying the introduction of special language support for referring to the request list.

Inheritance can be employed for reusing a sequential class carrying no annotations in the design of a modified class fit for usage in a concurrent setting. Non-standard scheduling can be added.

We noticed that Eiffel’s comment syntax is a minor technical nuisance for the annotations. We identified another weak point of Eiffel: the technique used for returning the result of a function - assignation to `Result` - is incompatible with postprocessing à la POOL.

So why Eiffel? The decisive argument was the availability of assertions and their integration with inheritance. We emphasized the close conceptual relationship between preconditions and guards and were able to associate delay semantics with an Eiffel precondition by mere introduction of the delay annotation. This approach is consistent with inheritance and the weakening of preconditions on redefinition.

5.4 Future extensions

Until now, we have not exploited another important assertion in Eiffel, the *class invariant*. One can easily envisage an invariant-related extension of the compatibility properties of a class towards "weak exclusion" of incompatible activities: when an activity is waiting for a reply, an incompatible request could be accepted *if* the invariant holds. This policy is a generalization of exclusion release at synchronization points in monitors (see also [Howard 76] [Haddon 77]). In languages without an integrated invariant concept, weak exclusion cannot be introduced in a safe way.

Concurrency control in databases is tied to the notion of *transactions*. In standard databases, transactions are usually required to be *serializable*. These notions are also useful in concurrent object-oriented programming, even if no persistence is involved. The specification of a class representing a subsystem implementation may postulate serializability in the following sense: concurrent invocations of an object of the class shall have the same effect as some serial execution of the corresponding operations - which play the role of transactions on the object.

Atomicity as introduced in 2.1 and 2.4 is only one of several possible ways of ensuring serializability. The implementor of a class may decide to achieve serializability with a carefully conceived concurrent implementation[8]. Yet another approach is automatic *two-phase locking*, preferably based on *semantic locks* (as opposed to read/write locks) on serializable component objects [Fekete et al. 89]. To support this, the compiler must know which operation pairs *commute* for each supplier class [Brössler/Freisleben 89]. Note that this information must be given by the specifier, not the implementor, of each supplier class. In accordance with our reusability goal, the commutativities should be expressed using *commutativity annotations*. Given these annotations for the supplier classes of an atomic class, a compiler switch would determine whether the class would indeed be compiled into atomic code or into concurrent code employing strict two-phase locking. Note that the implementor of the class is responsible for avoiding deadlocks in the latter case.

Serializable classes are built from serializable supplier classes. This nesting gives rise to *nested transactions* with possibly different concurrency control mechanisms on different levels. For each class, the implementor is free to choose any mechanism that seems appropriate; the decision is a purely local one.

It is important not to confuse compatibility (as introduced in 2.4) and commutativity. The former is an implementation property whereas the latter is a specification property. Two operations, e.g., `enqueue` and `dequeue` of a queue, may commute, yet their implementations may not be compatible.

A last remark is in order both for compatibility and commutativity. It is desirable that these properties may refer not only to operations but also to invocations, i.e., including the parameters (for commutativity in a persistence context see [Brössler/Freisleben 89]; for compatibility cf. the PO facilities mentioned in 4.4). This can increase the potential parallelism considerably and thus may be helpful on certain parallel architectures.

[8] There are examples for classes that *must not* be serializable and, consequently, *must not* be atomic.

Acknowledgements

I am grateful to Olaf Langmack, Jacek Passia, Irina Piens and Thomas Wolff for valuable comments on earlier drafts of this paper.

References

[Agha et al. 91] G. Agha, C. Hewitt, P. Wegner, A. Yonezawa (eds.): Proc. OOPSLA-ECOOP ’90 Workshop on Object-Based Concurrent Programming, Ottawa, 1990. ACM OOPS Messenger 2.2, April 1991

[Aksit et al. 91] M. Aksit, J.W. Dijkstra, A. Tripathi: Atomic delegation: object-oriented transactions. IEEE Software, March 1991

[America 87] P.H.M. America: POOL-T: a parallel object-oriented language. In [Yonezawa/Tokoro 87]

[America/van der Linden 90] P.H.M. America, F. van der Linden: A parallel object-oriented language with inheritance and subtyping. Proc. OOPSLA/ECOOP ’90, Ottawa, ACM 1990

[America 89] P.H.M. America: Issues in the design of a parallel object-oriented language. Formal Aspects of Computing 1.4, 1989

[Andrews et al. 88] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, G. Townsend: An overview of the SR language and implementation. ACM TOPLAS 10.1, January 1988

[Bahsoun et al. 90] J.P. Bahsoun, L.Feraud, C. Betourné: The "two degrees of freedom" approach for parallel programming. Proc. Int. Conf. on Computer Languages, IEEE 1990

[Brössler/Freisleben 89] P. Brössler, B. Freisleben: Transactions on persistent objects. Proc. 3. Int. Workshop on Persistent Object Systems, Newcastle, 1989. Springer 1990

[Caromel 90] D. Caromel: Concurrency and reusability: from sequential to parallel. JOOP 3.3, September/October 1990

[Caromel 91] D. Caromel: A solution to the explicit/implicit control dilemma. In [Agha et al. 91]

[Colin/Geib 91] J.-F. Colin, J.-M. Geib: Eiffel classes for concurrent programming. Proc. TOOLS-4, Prentice-Hall 1991

[Cook et al. 90] W. Cook, W. Hill, P. Canning: Inheritance is not subtyping. Proc. 7. Annual ACM Symp. on Principles of Programming Languages, 1990

[Corradi/Leonardi 91] A. Corradi, L. Leonardi: PO constraints as tools to synchronize active objects. JOOP 4.6, October 1991

[Decouchant et al. 91] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin, X. Rousset de Pina: A synchronization mechanism for an object-oriented distributed system. Proc. 11. Int. Conf. on Distributed Computing Systems, IEEE 1991

[Fekete et al. 89] A. Fekete, N. Lynch, M. Merritt, W. Weihl: Commutativity-based locking for nested transactions. Proc. 3. Int. Workshop on Persistent Object Systems, Newcastle, 1989. Springer 1990

[Gehani/Roome 88] N.H. Gehani, W.D. Roome: Concurrent C++: concurrent programming with class(es). Software - Practice & Experience 16.12, December 1988

[Gunaseelan/LeBlanc 91] L. Gunaseelan, R.J. LeBlanc: Distributed Eiffel: a language for programming multi-granular distributed objects on the Clouds operating system. Report 91/50, College of Computing, Georgia Institute of Technology, 1991

[Haddon 77] B.K. Haddon: Nested monitor calls. ACM Sigops OSR 11.4, 1977

[Herlihy 90] M.P. Herlihy: A methodology for implementing highly concurrent data structures. Proc. 2. Symp. on Priciples and Practice of Parallel Programming, ACM 1990

[Herlihy/Wing 90] M.P. Herlihy, J.M. Wing: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS 12.3, July 1990

[Howard 76] J.H. Howard: Proving monitors. CACM 19.5, May 1976

[ISO 90] ISO/IEC JTC1/SC21/WG7: Basic Reference Model of Open Distributed Processing. October 1990

[Kafura/Lee 90] D.G. Kafura, K.H. Lee: ACT++: building a concurrent C++ with actors. JOOP 3.1, May 1990

[Löhr et al. 88] K.-P. Löhr, J. Müller, L. Nentwig: DAPHNE - Support for distributed applications programming in heterogeneous computer networks. Proc. 8. Int. Conf. on Distributed Computing Systems, San José, IEEE 1988

[Meyer 88] B. Meyer: Object-oriented Software Construction. Prentice-Hall 1988

[Meyer 91] B. Meyer: Eiffel: The Language. Prentice-Hall 1991

[Papathomas/Nierstrasz 91] M. Papathomas, O. Nierstrasz: Supporting software reuse in concurrent object-oriented languages: exploring the language design space. In: D.C. Tsichritzis(ed.): Object Composition. Centre Universitaire d'Informatique, Université de Genève 1991

[Schmid 76] H.A. Schmid: On the efficient implementation of conditional critical regions and the construction of monitors. Acta Informatica 6.3, 1976

[Tripathi/Aksit 88] A. Tripathi, M. Aksit: Communication, scheduling and resource management in SINA. JOOP 1.4, November/December 1988

[Yokote/Tokoro 87] Y. Yokote, M. Tokoro: Concurrent programming in Concurrent Smalltalk. In [Yonezawa/Tokoro 87]

[Yonezawa et al. 87] A. Yonezawa, E. Shibayama, T. Takada, Y. Honda: Modelling and programming in the object-oriented concurrent language ABCL/1. In [Yonezawa/Tokoro 87].

[Yonezawa/Tokoro 87] A. Yonezawa, M. Tokoro: Object-oriented Concurrent Programming. The MIT Press 1987