Concurrency Annotations

Klaus-Peter Löhr
Institut für Informatik, Freie Universität Berlin
Nestorstraße 8-9, W-1000 Berlin 31, Germany
lohr@inf.fu-berlin.de

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

1 Introduction

Existing approaches to concurrent object-oriented programming suffer from several weaknesses:

1. Concurrency and synchronization do not blend well with inheritance.
2. Concurrency features (language constructs and/or library classes) are too low-level, sticking with traditional notions such as processes and inter-process communication.
3. Concurrent code is not reusable in a sequential setting although syntax and semantics often come close to a sequential variant. Vice versa, although sequential code often lends itself to a natural concurrent interpretation, this is usually not exploited.

3. is particularly annoying because it hampers concurrent software engineering, especially code reuse across the boundary between sequentiality and concurrency. This problem is present even with languages that are extensions of sequential languages, such as Concurrent Smalltalk [Yokote/Tokoro 87] or Concurrent C++ [Gehani/Roome 88].

Seen from the software engineering point of view, it would be attractive to take an integrated approach to the development of both sequential and concurrent object-oriented software: use *one* language which allows for both a sequential and a concurrent interpretation, depending on the compiler (or compilation switch) being used. Ideally, the sequential semantics of a piece of code should come "reasonably close" to its concurrent semantics.

OOPSLA'92, pp. 327-340

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 suggested 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 92].

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 significant, however, when interpreted by a compiler supporting a concurrent semantics. In addition, the concurrent interpretation of a given program text may be slightly different from the sequential interpretation even if no annotation is directly involved. The annotated version of the language is called *CEiffel*.

Sections 2 and 3 motivate and describe 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 4. Contention on access to objects raises scheduling questions, to be discussed in section 5. Our work on concurrent object-oriented programming is part of a larger effort to support the distributed execution of object-oriented programs in heterogeneous environments (project HERON). This context and the status of the project will be described in section 6. 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 Inter-object concurrency

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 any bias towards a 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] [LaLonde/Pugh 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. An atomic object will accept a request only when it is idle. We can think of two ways of how requests are generated:

1. A request is issued by another object through operation *invocation*. The originator of the invocation is called the *client* of that invocation, the invoked object is called the *server*.
2. An *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 also called autonomous.

A request that has been issued but not 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 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 to other approaches known from literature). Non-atomicity means *intra*-object concurrency whereas activeness causes *inter*-object concurrency. 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 speed-up, which of course depends on the parallel execution abilities of 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 im-

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

plies 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.compute1(y);--async.--
            compute2(z);
r.p; -- synchronization --
r.q;
```

may have weird effects if `server` or `y` are involved in `compute2`. 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. In Eiffel// asynchrony is explicit: it is provided by `PROCESS` objects only. Unfortunately, though, this ties asynchrony to atomicity.

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:

```
compute1(y: T1): T2 is --v--
do ..... end; -- compute1 --
```

Both the class and the objects are said to be asynchronous in this case. After an invocation of `compute1` the client proceeds immediately, even before the request is accepted[3]. Lazy synchronization is used in claiming the result, if any. - The annotation is ignored in local calls of the routine (i.e., calls from within the class).

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. Consider the alternative approach where a client uses a *fork* operation for introducing ad-hoc asynchrony. This leaves the client in control; but it is inappropriate in those cases where client and server have to *cooperate* to achieve a common goal which means that "interference" is mandatory rather than unwanted. Besides, declaring operations asynchronous allows for a more efficient implementation than ad-hoc forking. For these reasons 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.

In the absence of interference, the asynchrony annotation does not change the semantics of operation invocation. This is also true for many kinds of "weak interference" based on commutative operations on shared objects. As pointed out above, it cannot be upheld for all cases of interference.

[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 --
```

Under concurrent interpretation, an autonomous routine generates requests for itself as mentioned under 2.1.2. If it is exported, additional requests may be generated through invocation by clients (which proceed immediately, as with an asynchronous routine).

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;
   stepTime: Real;

   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 3.1 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`).

Asynchronous and autonomous operations offer several advantages over more traditional concepts such as a "body" describing the lifelong behaviour of an active object (as in Concurrent C++ and Eiffel//; see also POOL [America 87] [America 89] and ABCL/1 [Yonezawa et al. 87]). 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.

Consider a non-autonomous version of the class Moving where step would be exported and the steps of a moving object would have to be driven by an external force. Such a class is easily upgraded into an autonomous version. Vice versa, the autonomous version is readily reusable in a sequential environment.

Note that an autonomous class can be *specified* just like a sequential class, only with additional mentioning of which operations are autonomous. A client has to be aware that the server can undergo spontaneous state changes, as if "daemon" clients causing these changes were present.

2.4 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--).

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 whether the call c.op (with c of type C) will generate a synchronous or an asynchronous activity. Thus, the situation is safe only if A's implementation of op is interference-free (2.2).

3 Intra-object concurrency

3.1 Compatibility

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;
```

Compatibility has nothing to do with the specification of a class. The 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.

A finer granularity of the compatibility relation can be achieved by interrelating *requests* rather than operations. This takes the parameters into account, as in

```
update(k: Key; d: Data) is
        --|| update(x,y) if x/=k,
             lookup(x)   if x/=k --
```

It causes more overhead but can increase the potential parallelism considerably and thus may be helpful on certain parallel architectures. The syntax of an individual element in a compatibility annotation is

Unqualified_call [**if** Expression]

As a central property of CEiffel, compatibility annotations *are independent of* asynchrony and autonomy 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). This implies the strict independence of intra-object concurrency and inter-object concurrency as postulated in 2.1.

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 *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*.

3.2 Controlled objects

Within one program, the annotations in a class are observed or ignored on a per-object basis. Whether an object is going to behave according to the sequential or the concurrent interpretation of its class is determined by the

control annotation `--!--` .

The annotation is attached to the declared type of an entity, as in

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

It affects the creation of `q`, declaring that the object[4] is to be *controlled*, i.e., that the annotations are in effect. In addition to object-specific controlling, there is also class-specific controlling: `--!--` can be attached to a class head; this causes all objects of that class to be controlled.

[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.

If a class is not fully concurrent, the control annotation causes a *concurrency control* scheme to take effect. 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).

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); for details see section 5.

Object-specific controlling is rare. Most objects of user-defined classes 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. The usage pattern of a shared object ultimately determines whether control is necessary or not.

The basic classes `Integer`, `Boolean` etc. are atomic and controlled; this is implemented in hardware (indivisible read/write operations). The library classes `Array[T]` and `String` are atomic but not controlled.

3.2 Compatibility and inheritance

The properties of an object are determined by its class, but also by all ancestors of that class. Thus, if inheritance is involved, the adjectives *atomic* or *concurrent* do not necessarily carry over from a class to its objects.

If class `B` *inherits* from class `A`, the text of `B` may include compatibility annotations referring to inherited routines. A *redeclared* routine may be annotated differently from the original. With *multiple inheritance*, any routine inherited from one parent is compatible with any routine from any other parent (unless we have repeated inheritance with sharing, or deferred routines are involved).

Remember that all classes implicitly inherit from the universal class `Any`. This class provides routines for cloning, copying and comparing objects; reading of entire objects is involved here; copying also involves writing. Reading an entire object is compatible with itself and with all other operations explicitly or implicitly annotated with `--||....--`. No compatibilities are declared for writing an entire object.

The compatibilities introduced by declaring an heir concurrent (attaching `--||--` to its head) do involve *all* inherited operations. Objects of such a class are fully concurrent, i.e., there is no concurrency control.

4 Guarded classes

4.1 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) asynchronous we could do so, but, as mentioned in 3.2, 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 re-evaluating 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 3.2.

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.

4.2 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:

```
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 between two assertion clauses in a precondition [5]. It divides the precondition into two parts, the *checker* and the *guard*. The checker of an autonomous routine must be empty. An object invocation that violates the precondition causes an exception if an assertion clause in the checker is violated; otherwise, a delay occurs. A class that carries guards is called a *guarded class*; its objects are called *guarded objects*.

Several preconditions in the above examples have to be annotated using --@-- . A variant of enqueue might read

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

The precondition of the first version of the deposit operation in Repository may be written either

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

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

or

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

depending on the desired semantics.

4.3 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`.

5 Scheduling

Pending requests raise the question of how the acceptance of requests is to be scheduled. The default scheduling strategy is basically FCFS: acceptable requests are accepted in the order they were issued. This strategy prefers a pending request, as soon as it becomes acceptable, over a new acceptable request. It cannot, of course, prevent indefinite delays.

We give a precise description of the default scheduling strategy. Associated with every object that is controlled or guarded (or both) is a *request list* which at any time contains the pending requests for that object. A pending request is much like a variant record, with the operation corresponding to the variant, the formal arguments to the record fields and the actual arguments to the actual record components.

When a request arrives at an object it enters the request list. If the compatibilities allow a corresponding activity to be started, the activity is tentatively started, evaluating the checker. If the checker is not satisfied, the request is removed from the request list, the activity is aborted and an exception is raised. In any case, the client is allowed to continue. Then the guard is checked. If it is satisfied, the request is removed from the request list and the activity continues (acceptance); if not, the activity is aborted and the request remains pending (delay). When an activity terminates, the pending requests are scanned in arrival order, and each request is treated just as described for an arriving request. The following pseudo-code gives a slightly more precise description. Activations of `request arrival` and `activity termination` are executed under mutual exclusion.

```
request arrival:
  enter request list;  check .

activity termination:
  for each pending request(FCFS)
  do  check .

check:
  if compatible then
     if checker ok then
          if guard ok then
          remove from request list;
          accept    end
     else remove from request list;
          raise exception end end .
```

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 leaves the realm of a sequential programming language. Annotations cannot do the job any more. We take an approach that is based on read-only access to the request list. This allows to refer to pending requests either in preconditions or in special scheduling routines and blends well with inheritance. - A detailed discussion of explicit scheduling is beyond the scope of this paper; the reader is referred to [Löhr 91].

6 Context and perspective

6.1 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 an 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 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].

6.2 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 object interaction is implemented as procedure call.

2. Atomic active objects: One thread is used for all activities of an object, and object interaction is implemented as message passing, possibly across address space boundaries.

3. Folding atomic active objects: If there is a plentitude of objects of the same class (think, e.g., of a video game) and the underlying architecture is not highly parallel, all activities of those objects could be handled by one thread.

4. Chaining asynchronous activities: If an asynchronous activity ends with the invocation of another asynchronous operation, the same thread can be used [Löhr 92].

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 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 (this is important for multi-processor architectures).

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.

6.3 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
`--||...--` : compatibility
`--@--` : delay on assertion violation
`--!--` : controlling

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.

We noticed that Eiffel's comment syntax is a minor technical nuisance for the annotations. We identified another weak point in Eiffel: the technique used for returning the result of a function - assignation to `Result` - is incompatible with post-processing 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.

Acknowledgements

Thanks go to Olaf Langmack, Jacek Passia, Irina Piens and Thomas Wolff for valuable comments on earlier drafts of this paper. Olaf also contributed to this work by testing Eiffel-3 code on the Eiffel/S system.

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

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

[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

[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 multigranular distributed objects on the Clouds operating system. Report 91/50, College of Computing, Georgia Institute of Technology, 1991

[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

[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

[LaLonde/Pugh 91] W. LaLonde, J. Pugh: Subclassing ≠ subtyping ≠ is-a. JOOP 3.5, 1991

[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

[Löhr 91] K.-P. Löhr: Concurrency annotations and reusability. Report B-91-13, Fachbereich Mathematik, Freie Universität Berlin, November 1991

[Löhr 92] K.-P. Löhr: Concurrency annotations improve reusability. Proc. TOOLS USA '92, Santa Barbara. Prentice-Hall 1992

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

[Meyer 92] B. Meyer: Eiffel: The Language. Prentice-Hall 1992

[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