SERIE B – INFORMATIK

The moderate approach to integrating concurrency and object-orientation

Thomas Wolff
wolff@inf.fu-berlin.de

B-95-07
May 1995

Keywords

language constructs, mixing paradigms, concurrency, objects

Abstract

Principles of integrating concurrent computation, objects, and inheritance are discussed. The approach outlined here is guided by general considerations about the rôles of the two paradigms being merged. We suggest our approach to some open design questions: the relation of concurrency to inheritance and the thread and synchronisation concept among objects. The question, where should the combination of concurrency and object-orientation be settled between the paradigms, is analysed in different aspects. The presented approach also avoids the inheritance anomaly.

Institut für Informatik
Freie Universität Berlin
Takustraße 9
D-14195 Berlin

0. Contents

As an extended introduction, the first three chapters of the paper, "Concurrency and object-oriented programming", "Concurrency and inheritance", "Concurrent computation models and distribution in the object context", roughly present usual models to approach these topics, as well as first hints on why our approach deviates from them. The central chapter, "A moderate integration approach", presents our model along five basic criteria, again discussing its justification against other concepts. Finally, we summarize our approach and outline some future directions for its development.

1. Concurrency and object-oriented programming

For placing concurrency and object structure together in one programming language, two mainstream approaches can be separated by their motivations:

- The *object-founded approach* wants to enrich object-oriented programming with the presence of concurrency. It often exhibits processes attached either to operation invocations (as in the guardians of [LisSch]) or to the objects themselves (as in [Emerald] or [Caromel]), or else very specialized objects evolve which combine asynchronism with small scale behaviour (such as the actors of [Agha]). The object-founded approach often takes it for granted that the object-oriented view conquers the world of software and languages. In the overview article [Nelson] we find the general remark "A process is essentially just another object ... Therefore, the class mechanism ... provides a good starting point for creating processes." But since "the world is not all objects" ([Goguen]) our argument will be that this is not a good starting point at all.
- The *concurrency-founded approach* wants to extend concurrent programming into object-oriented environments. It is usually quite liberal with placing processes into objects (for example [SR] or [Trellis]) and tends to make use of explicit, rather low-level synchronization mechanisms such as semaphores and queues.

Concurrency as a property of problems and as a paradigm of program development is an important aspect of language design. The concurrent structure of a problem or an algorithm may in some cases depend on the structure of the data involved. In other cases it may be inherent to the algorithm and independent of the data structures. Programmers need the freedom of choice with respect to which approach or programming methodology is most appropriate for their problem. For this reason, we claim that the integration of concurrency with object-oriented languages should not be performed from the object-centred point of view. Any model of "concurrent objects" which sets them up as something new and special (the object-founded approach), imposing on the programmers to permanently "think in terms of objects" when programming concurrent algorithms, is too restricted to suit the requirements of concurrent programming. On the other hand, a completely orthogonal model (the concurrency-founded approach) would not be appropriate either, for the given object structure comprises a principle of encapsulation and integrity that must be respected by other mechanisms present in the language. We need to find a combined approach which is related to – but not focused on – object structure, and also related to the idea of algorithmic concurrency, each in its respective aspects.

2. Concurrency and inheritance

Potential problems with multiple inheritance[1] strongly influence a group of design decisions which we explain with the most basic one, the granularity of concurrency.

Our model of concurrency in objects will include multiple threads of control inside any object (in contrast to just one). This is not only a design idea motivated by the concurrency-related aspects of our approach (especially the thread concept, see below), but it also respects best the inheritance structure of object systems, especially in the presence of multiple inheritance. "Proof": Suppose we had the restriction to at most one activity thread in an object – this might be a process attached to (or even identified with) the class as a source of algorithmic concurrency, or, like in [Caromel] or [POOL], a specialized activity needed to express explicitly programmed concurrency control. In most object-oriented languages, objects are instances of their static description, the classes. So we would have to define the activity of the objects of a class within that class. Now for multiple inheritance we would either have to exclude activity from being inherited, which would appear somewhat artificial and would impose on the programmers the need to renew any synchronization-related special code (*the* object activity in the languages mentioned above) in inheriting classes. Or else, we would immediately have a contradiction with our assumed single-thread restriction as suddenly all inherited activity specifications would show up in our newly defined base class, demanding that our concept provide some special rule to solve the conflict. Since such a rule would not be introduced for its own quality but as a mere remedy for a problem, it would be weakly motivated and probably rather arbitrary. In general, such non-productive rules do not contribute to the quality of a language.

Basically the same sort of conflict would arise with any special property that classes might have or not have, like e.g. the monitor property that disallows multiple simultaneous activity in certain classes (if this rule is maintained for *all* classes, we have a "quasi-concurrent" object language according to [Papathomas]). So all class-related behaviour

1 Since multiple inheritance is an elementary feature for structuring object-oriented programs we do not consider the resort of restricting languages to single inheritance.

rules (especially those concerning concurrency) that might be thought of, would confront us with semantic problems:

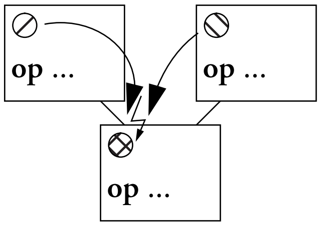

Figure 1: Incompatible class properties, ⊘ and ⦸ , are attached to two superclasses of a multiple inheritance hierarchy, yielding a conflict in the subclass if inherited: ⊗ .

We do not think any solution constructed to remedy the conflict in this situation is motivated other than for getting rid of the problem. Therefore, we prefer to avoid the problem from the outset.

The answer how to achieve this is simple: No class-based special properties and no single-activity restriction admitted! The latter implies that we have to design a concept to cope with the problems of multiple simultaneous activities inside objects in a way respecting the object structure. Especially, mutual activity exclusion properties have to be definable on a more specific, operation-related basis.

This is also part of our answer to the *inheritance anomaly*. Using this notion, [MatsYone] address the inheritance conflicts that arise with any means of synchronization attached on the class level, e.g. explicit code for synchronization control or global synchronization specifications like path expressions.[2] [Papathomas] also points out the problem. We do not only avoid synchronization control code at all but also the synchronization mechanism that we suggest will be compatible with inheritance in that it cooperates symmetrically within an inheritance hierarchy so that previous (i.e. inherited) synchronization specifications can be referred to and combined with new ones without the need of redefining them. This will be detailed below.

If we consider the mechanisms that lead to the inheritance anomaly, we find that especially those approaches that strive for an object-founded integration of concurrency give rise to the problem. In contrast, it is in the spirit of the concurrency-founded approach to combine concurrency features with object structure more freely. So it is one of the core concepts of object-orientation itself (i.e., inheritance) which leads the object-founded approach to integrating concurrency to a dead end.

3. Concurrent computation models and distribution in the object context

A section about this topic is called for because often, the topics of concurrency and distribution are treated in conjunction with each other. This is mainly because one of the

2 [Meseguer] presents a solution to the inheritance anomaly but resorts to a completely different style of programming to achieve it.

motivations for distribution, the increase of performance, only makes sense if it is accompanied by concurrent computation. On the other hand, distribution and concurrency can both live independently from each other; some classical RPC-based (modular) distribution systems do not provide concurrency themselves (e.g. [BirNel]); also our own distribution project [Heron] shows that an object distribution system maintaining full concurrency compatibility can be established even with only marginally taking account of specific concurrency details.

We'd rather separate these topics, and handle them as orthogonal issues, especially since, in the author's opinion, their respective relation to the task of programming and thus their desirable degree of affecting the programming language model is completely different. Whereas concurrency is basically a paradigm of expressing algorithms, and should therefore be explicit, distribution is predominantly a paradigm of exploiting resources, and should therefore be as implicit as possible.

Depending on the motivation of combining concurrency and object-oriented system structure, and the degree of having distribution in mind, varying approaches to the concurrent computation model can be identified:

- In a plain concurrency-founded approach, explicit concurrent computation is orthogonal to the object structure. Additional features for distribution may be present where distribution is handled explicitly. Synchronization of concurrent threads is organized by conventional means independent of the object structure. Objects are passive entities with respect to the concurrent thread system.
- In a distribution-founded approach, often the desire for increased computational throughput calls for asynchronous remote operations. They may be explicit in a more conventional RPC-like environment, or they may provide implicit concurrency. Either way, concurrent computation comes onto the objects by invoking their operations asynchronously, leaving the objects still passive.
- In an object-founded approach, if the idea is to attach concurrent computation to objects, more probably some close connection between objects and processes would result. All objects, or objects of certain classes, are active maintainers of concurrent threads which in return usually do not leave the realm of their object. For the communication between objects, messages or calls between threads may be part of such a concept.

No matter which reason or motivation is prominent, computational concurrency models for objects (which tend to correlate with the two opposite approaches) have so far seemed to be basically one of two extreme point solutions:

- In the passive object model – predominant in the concurrency-founded approach –, processes are independent entities which do not reside in but only pass through objects.
- In the active object model – predominant in the object-founded approach –, contiguous activity (called process or thread) is stuck in one object, employing different means of interaction and communication between processes or objects.

This classification can also be found in Papathomas' survey. In accordance with our design principles outlined above we think that both ideas focus too much on the notion of either object or process. The reservations against both sides can be identified more clearly by separating details of the interplay of concurrent computation and object structure as shown in the following chapter. We also show that a solution doing justice to both programming paradigms need not be a mere compromise.

4. A moderate integration approach

As indicated, we do not favour one of the border solutions, dominating one programming paradigm by the other. The author is especially convinced that it is a fashionable myth to view object-orientation as a super-paradigm capable of subsuming all the others.

Instead, a middle course should be found which treats both paradigms as equal partners and leaves the essentials of both in the result. With this in mind, we consider five basic design questions for integrating concurrency and object-orientation and present the answers that comply with our intent of a moderate approach:

(1) Where does concurrency occur?
Or: What is a thread? How is it related to objects?

(2) What are the sources of concurrency?
Or: What are the possible origins of threads? At which places in the program can threads be created and by what means?

(3) Which methods of synchronization do we maintain?

(4) How do we schedule?

(5) Which special problems do arise combining concurrency and inheritance?

We will now discuss our concept's approach to these design questions and contrast it with more biased approaches where appropriate.

4.1. Where does concurrency occur?

Concerning the CONCURRENCY-FOUNDED APPROACH, on the one hand, we are not going to accept complete independence of threads from the object structure of programs – interaction between threads must obey certain integrity rules attached to the objects. This can be regarded as to only mean accepting an object's integrity as a data encapsulation unit such as a database record, a data structure, or an integer. This means that mutual exclusion rules for concurrent threads must be specifiable and have to be realized on a problem-specific basis (for example, exclusive vs. shared access for certain data or operations). This leads us to question (3) below about synchronization but does not force us to restrict the general concept of threads as such.

Concerning the OBJECT-FOUNDED APPROACH, on the other hand, we are not going either to stuff the complete concept of concurrency into single objects alone. That approach would detract from the more general idea of concurrent computation: parts of a program being formed by multiple threads of contiguous activity over a set of data and procedures. Upholding this generality is motivated by the wish to supply the most suitable features for problem-oriented programming: We do not want a new special concept of concurrent objects but rather we want to supply concurrent programming capability in the object environment.

As a result, in our model – the MODERATE APPROACH – we will have both intra-object concurrency (among operation invocations) and inter-object concurrency as a consequence of concurrent activations of objects as well as autonomous creation of activity (see below).

Moreover, to come to a clarification of the concept of a thread or process, we claim that the prevailing restriction of threads in object systems – whether with or without intra-object concurrency – to reside in one object only, while an invocation of another object automatically means spawning of a new thread (and possibly suspending the calling thread), serves no semantic purpose. This view is only taken to be object-oriented and might be induced by the misleading notion of message passing[3]. We think our notion of a thread is a more natural one: *A thread is created by the start of the program or the execution of any thread-originating language construct as listed under item (2) below; apart from further new threads created by such constructs, it extends over all invocations of operations caused by it, regardless of any object boundaries passed.* This does not only define a notion but also has semantic significance for synchronization as explained below.

DISCUSSION: Some approaches reject intra-object concurrency or promote the implicit serialization of concurrent uses of an object, making their behaviour similar to monitors. This restriction might be influenced by viewing objects as elementary units of activity or as implementations of automata. But it complies neither with a more general view of objects as encapsulations of data and algorithms, nor with the principles mentioned above, i.e. most general and unspecific integration of concepts. On the other hand, inter-object concurrency is the basic expression of concurrency in an object environment since it relates objects to each other concurrently. The concept presented here has room for both.

4.2. What are the sources of concurrency?

Actually, our general attitude does not strongly bias any special choice in this respect apart from a guideline: Thread creation is an original aspect of concurrent programming and should therefore be organized from the concurrency point of view. Hiding it in the object context, e.g. via asynchronous operation invocations, is not considered appropriate as an implicit and sole source of concurrency; but it is alright as an optional alternative.

On the basis of our thread concept, conventional explicit thread creation is perfectly compatible with object-orientation. Essentially, all of the thread creation mechanisms listed below are acceptable in the object-oriented context. A choice among them (or even the decision to admit them all) is basically a question of what forms of expressing concurrency are desired, not how much concurrency is consistent with the object model.

3 usually meaning nothing more than procedure or function call

The list will be terminated by the mechanism that appears most special to object orientation, the *autonomous operation*.[4] But even this one has its analogy in static process declarations in languages like Ada or SR ([SR]).

For notation, let us assume that the declaration of a normal object operation (i.e., a procedure or function)[5] looks like

op p (*parameter specification*) *optional result specification* **is**
do *operation body* **od**

Our concept has room for the following different methods of thread origination:

- Static declaration of concurrent threads:

  conc thread_1 || ... || thread_n **cnoc**

  runs n groups of statements as parallel threads, while the surrounding thread waits for their termination.

- Dynamic creation of a concurrent thread:

  fork thread **krof**

  runs a group of statements as a new parallel thread, while the surrounding thread continues.

- Static declaration of asynchronous operation invocation:

  async op p (...) **is**
  do ... **od**

  declares the operation p to always be invoked asynchronously as a new thread.

- Dynamic asynchronous operation call:

  spawn p (x) **spawn** objectx.p (x)

  invokes an operation asynchronously as a new thread.

- Declaration of autonomous operation invocations:

  auto op p **is**
  do ... **od**

  declares the operation p to be an *autonomous operation*. It gets invoked without external cause. Its unsolicited invocation may be scheduled by the system at arbitrary points of time during program execution unless restricted by further mechanisms. (Restrictions will normally be added to yield more specific and thus useful sets of invocation occasions, e.g. restricting p against multiple simultaneous invocations of itself; see in 4.4, "How do we schedule?".)

Why does this set of features, especially including autonomous operations, comply with our integration philosophy while other mechanisms which might on the first sight only have subtle differences do not?

4 With respect to thread origination, this construct is related to the similar basic construct in [Löhr].

5 We do not adopt the wide-spread but queer terminology speaking of *methods* where ordinary procedures or functions are meant, and of *messages* where calling them is meant.

- There is no implicit concurrency semantics attached with the event of communication between objects. Every concurrency-related semantic property is tied to single operations instead.
- A thread of activity, even if originated by an autonomous operation invocation, is not tied to the object that originated it. It can spread over all objects of the whole program.
- Autonomous activity is not *the activity of an object* but *an activity within an object*. There is no restriction of the number of autonomous operations any class of objects can bear and the presence of autonomy within a class of objects does not mutate it into another kind of class[6]. Also, there are no additional programmer-supplied precautions necessary in objects equipped with autonomous operations.

4.3. Which methods of synchronization do we maintain?

The *critical region*, which provides for mutual exclusion, can be regarded as a basic control structure for structured synchronization.

We do not think the use of more elementary synchronization elements (e.g. semaphores) is appropriate in the context of high level languages supporting systematic program development.

On the other side, even the free placement of critical region synchronization blocks in a program still seems extremely liberal in a language structured by the object model. Therefore, we will only admit specifying synchronization means in a restricted form: In our concept, object operations can be annotated with *entry conditions* that include exclusion specifications, and moreover these do not define synchronization on explicit regions but exclusion against certain other operations as this operation's *exclusion goals*:

op p (...) **exclude** q_1, ..., q_n **is**
do ... **od**

> specifies that no simultaneous invocations of the operation p and any of the operations q_i by different threads will be allowed. q_i = p is a legal and perfectly reasonable special case. As opposed to some exclusion region which would have to be declared seperately, the operations q_i are *exclusion goals* of the exclusion specification attached to the operation p. An exclusion goal q_i is called *open* for the current thread if the condition above holds for it, i.e. no other thread is currently performing within the operation q_i including any sub-computation.

In general, entry conditions might depend on exclusion goals, Boolean conditions (*guards*), or communication availability. The latter does not apply to our concurrency model since we do not have explicit message reception. The first two kinds of synchronization conditions shall be permitted in free combination, i.e. multiple exclusion goals (as shown above) and a Boolean expression, wherever synchronization may occur:

op p (...) **exclude** q_1, ..., q_n **when** B **is**
do ... **od**

6 We could call objects that have autonomous activity *subjects* so as to introduce a nice notion.

adds a Boolean expression B as a guard to the operation p. The guard B is another kind of *entry condition* which is called *open* if B evaluates to TRUE.

We can now summarize the effect that entry conditions have on a thread which wants to perform an operation which has been called. Entry to the operation p is allowed only if all entry conditions are open as defined above. Moreover, it is an important requirement that the determination of the entry conditions must be *atomic* in effect, i.e. the system has to guarantee that upon actual entry of the operation's body all of the entry conditions are still open (i.e., all exclusion goals q_i are open for the current thread and B is TRUE). At this very moment, p itself stops being open as an exclusion goal for other operations.

We say an invocation of an operation is *enabled* if all its entry conditions are *open*.

4.3.1. Avoiding reflexive deadlock

As a special problem of mutual exclusion apparently not identified previously in the literature, we consider the following situation:

Assume a thread of control acquires an item of exclusion, i.e. synchronizes on an exclusion goal which thereby gets closed (e.g. enters a conventional exclusion region or blocks a semaphore). During that time the same thread may request the same exclusion goal again (this may well happen from another place of execution / another object; it may as well be the result of recursion). Conventional exclusion mechanisms, working locally, would just block the requesting process and thus cause a *reflexive deadlock*, i.e. the thread would lock itself into a deadlock. This problem, though probably not occurring naturally in specialized applications such as operating system related ones, must not remain unconsidered for a general purpose language. We think it is not appropriate to accept this sort of deadlock from a programming systematic point of view. We have already included semantic precautions to avoid the problem. The observant reader will have noticed that the attribute "by different threads" is attached to the requirement that no simultaneous invocations of the operation to be synchronized and an operation of an exclusion goal is allowed. This means that if that exclusion goal is being hold by the current thread it will still be regarded open and the thread may pass into the operation in question. (If the synchonization mechanism was an exclusion region, this would mean the thread may enter the same region again.) This semantics is also the reason why a concept of a global thread, spanning all cross-object (and also distributed) computations, is important.

4.3.2. Considering alternative exclusion methods

— *Releasing the exclusion during sub-computation.*

In the case of the nested monitor discussion ([HADDON]), the concept of weak monitors was proposed, which releases the exclusion during sub-computation outside of the monitor. This did not aim at the reflexive deadlock problem but rather at the problem of maximizing resource availability of the monitor module for any other client. However, this answer would neither cover the case of direct recursion nor do we consider it acceptable at all since it breaks the purpose of an exclusion goal, assuring computational integrity of data at a fairly abstract level. Also, by our finer-grained operation-related synchronization concept, the problem of resource availability is much smaller than with the coarser monitor concept.

— *Explicit synchronization control.*

We have already pointed out (in chapter 2, "CONCURRENCY AND INHERITANCE") that this mechanism belongs to those conflicting with multiple inheritance, this one even being addressed by a special notion in the literature, "inheritance anomaly" ([MATSYONE]). We would consider anyway that active synchronization control which is placed apart from the actual operations to be synchronized is a fairly technical mechanism and thus rather confusing in the course of easy construction of software as the author thinks should be assisted by language design. Therefore we are not at all unhappy that this concept is already precluded by the "no class properties" principle set up above.

— *Specifying a unique exclusion semantics for operations.*

This approach to avoid explicit synchronization would in effect result in something similar to monitors. (Having to avoid class properties, it could still be defined as the language-immanent exclusion property of *all* classes.) However, we think the monitor concept is a very special solution especially suited for operating system related programming tasks. It may often be a useful primitive there but does not provide the desirable flexibility for a general purpose language. Concurrent programming needs in objects with synchronization requisites would be arbitrarily constrained. Therefore we do not approve of this alternative. After all, the author does not think that being explicit would be a disadvantage in concurrent programming if only the language features of explicit expression are sufficiently abstract.

— *Automatic synchronization based on semantic requirements.*

In order to exploit maximal parallelism while not having to care about synchronization at all, we could try to derive the required synchronization measures from the operations' specifications and bodies (in Eiffel, e.g., the specification could be taken from pre- and post-conditions). It would then be necessary to analyse the interrelations between operations at compile-time and probably also to employ a flexible synchronization concept such as transactions. It is unclear at the moment what can be achieved by such analysis; moreover, the integration of transactions would leave the imperative execution model underlying usual object-oriented languages. Therefore we do not currently view this as a feasible and desirable approach to combining concurrency with a normal object-oriented language.

4.4. How do we schedule?

4.4.1. Explicit scheduling on priority and conditions.

For many concurrent algorithms it is desirable to express operation entry conditions to depend on a more global state of the computation, including not only variable contents but also pending operation invocations. So some feature to take them into account must be devised. Since we reject explicit separate control, this has to be integrated into the entry conditions attached to the operations in a rather declarative way.

First, we allow the Boolean entry condition to refer to parameters of the operation invocation. Second, we extend the entry conditions by two Boolean sub-expressions which can refer to the state of other operation invocations waiting for their activation:

all p: B

> is a Boolean expression with p being the name of an operation and B a Boolean sub-expression in which for every formal parameter x of the operation p the notation p.x may be used to refer to the actual parameter of an invocation of p. The expression is TRUE if for *all* pending invocations of p the sub-expression is TRUE with the respective values of parameters inserted.

some p: B

> (Replace "*all* pending invocations" above with "*some* pending invocation".)

EXAMPLE:

op p (x: integer) **when** x < xx **and all** p: p.x >= x **is**
do ... **od**

As shown in the example, the **all** clause can be used to schedule among pending invocations by comparing their parameter values against those of the current invocation. In case direct scheduling by an integer expression is considered more useful, an alternative or additional priority expression (like in [SR]) could be employed:

op p (...) **by** i **is**
do ... **od**

> as a abbreviating equivalent of

op p (...) **when all** p: p.i >= i **is**
do ... **od**

4.4.2. Activation of autonomous operations.

The activation conditions for autonomous operations have been left rather unspecific so far. Obviously, our definition that their invocation may be scheduled "at arbitrary points of time" by the system will not be very useful in most practical cases. However, we already have a good mechanism at hand to constrain this arbitray non-determinism. Usually, the declaration of an autonomous operation will list a number of exclusion goals including itself, thus relieving the scheduler from the burden of discretion about how many and how frequent multiple invocations of an autonomous operation would be appropriate.

In contemplating about further mechanisms for controlling autonomous operations, we had best consider what purposes autonomous operations may serve:

- Cyclic autonomy

  Regular behaviour of the object like pseudo-continuous change of state or repeated activity could give rise to approximate real-time behaviour desired with autonomous operations. Some time interval could be used to specify this with the declaration:

  auto op p **every** n [**milli**]**seconds is** ...

  specifies that regurlarly at the given time interval an invocation of the autonomous operation will be issued. Its actual activation is still subject to further synchronization specifications and general scheduling.

  Example:

  auto op movement **every** 10 **milliseconds is**
  do distance := distance + speed / 100 **od**

- Tidying-up autonomy

  Clearing up the local state triggered by the execution of certain other operations could be achieved by various means. Explicit enabling of an autonomous operation (like the **enable / disable** operators in [LOGLAN]) would be a weak non-determinate form of triggering and can already be simulated by a simple flag anyway. Explicit launching of an invocation would be more suitable to achieve a functionality similar to a rendezvous ([Ada]) or a post-processing section ([POOL]), but would be covered by a concept of asynchronous calls.

  If we put it the other way round, having the autonomous operation specify which other operation(s) it is a *post-processing operation* for, we have a new feature with additional, advantageous functionality: An autonomous operation can be specified for post-processing of an operation in a superclass in which this relation had not yet been devised.

  auto op p **checking** q_1, ..., q_n **is** ...

  specifies that operation p gets enabled and subject to scheduling whenever an invocation of one of the operations q_n terminates.

- State-dependent autonomy

  Tidying up or keeping the object in an approximate state of usability can sometimes also be useful on a more sporadic basis, depending on changes in the object's state. This can already be expressed with the Boolean entry conditions.

  Example 1:

  class stack ...
  auto op keepenoughbufferspace **when** remainingbufferspace < 15 **is**
  do getsomemorebufferspace **od**

  Example 2:

  auto op garbagecollect **is**
  do findsomegarbage; disposeit **od**

4.4.3. Underlying scheduling strategy.

With respect to the Boolean conditions and exclusion goals that define operation invocations to be enabled or not, and regarding the fact that invocations of autonomous operations can occur spontaneously, scheduling among all enabled operation invocations should be *fair* (in an appropriate sense not defined exactly here, cf. [Francez]). Although this may slightly complicate the system's scheduling strategy, we approve of it because we believe fairness is an important concept for realizing the programmers' intention in supplying several operations of equal rights. (We must admit, though, that we have not solved the DWIM[7] problem either.)

4.5. Which special problems do arise combining concurrency and inheritance?

Inheritance and class properties: This important question was already addressed above in the chapter "Concurrency and inheritance". Having argued that the only natural solution to the problem of contradicting inherited properties is disallowing class properties at all, we also laid the ground for certain other design decisions like the unacceptability of a restriction to single-activity inside objects. This means we have "concurrent process structure in objects" in terms of [Papathomas]. All concurrent activity is simply inherited, just as other object components. So we also do not need to redefine *the* object activity as must be done in [Caromel].

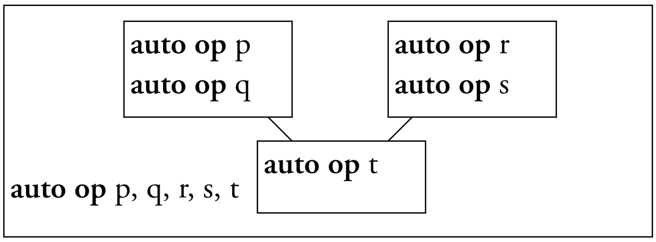

Figure 2: (Example) With two autonomous operations in each inherited class and another in the subclass itself, a total of 5 autonomous operations shows up in the new class.

Inheritance and the synchronization mechanism is a more sophisticated topic. Synchronization properties of superclasses are inherited. The exclusion goal mechanism was designed such that its exclusion effect is symmetric in the sense that the exclusion semantics of the following three pairs of declarations is exactly the same[8]:

op p **exclude** q **is** ...	**op** p **is** ...	**op** p **exclude** q **is** ...
op q **is** ...	**op** q **exclude** p **is** ...	**op** q **exclude** p **is** ...

The advantage of this is that exclusion goals can refer to operations of an inherited superclass and can set up mutual exclusion properties even if this was not devised previously in that superclass. So to apply our synchronization mechanism in its full

7 Do What I Mean.

8 Our implementation concept, which bases exclusion goals on implicit exclusion regions, shows that this semantics is fairly simple and efficient to achieve; cf. remarks on it below.

flexibility there is no need of redefining superclass synchronization. Thus we do not get into trouble with the "inheritance anomaly".

Inheritance and synchronization usage is primarily a programming problem. Suppose two inherited classes use variables of a common superclass for which each of them did not need to specify exclusion for itself. In a common subclass, this need may arise due to increased degree of concurrency. In this case, the superclass supplying those variables, or the intermediate classes using the variables, may have to be modified additionally.

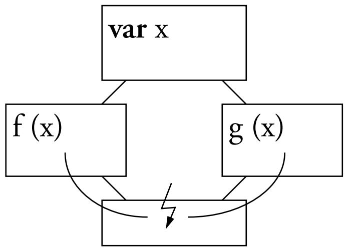

Figure 3: Although in objects of each of the two middle classes the use of the variable x may not require any synchronization measures, this might be necessary in a further subclass which combines these uses of x concurrently.

However, this is not a special problem of synchronization. Modifications or extensions to inherited code can never be precluded completely since there is no perfect universal solution to the problems of the principle of reuse in software development. Therefore we do not think special language provisions concerning synchronization in this case are called for (i.e. the task of resolving the problem is left to the programmers).

5. Concept characteristics

We have considered which principles should be observed so that the language paradigms concurrency and object-orientation get together in a harmonic manner. Further we have outlined a model which does justice to both.

As a consequence of our moderate approach to finding integrating principles, the resulting concept of language constructs and properties reveals, within its respective parts, different degrees of connection of the concurrency and object concepts:

- The basic concept of the existence of activity of threads is object-independent. Once created, a thread can cross object boundaries, keeping its identity. This is also an important point in preventing reflexive deadlock.
- In addition to conventional means of thread creation (**conc** / **fork**) that could be incorporated, our special means of thread origination, the *autonomous operation*, is object-related. This does not mean, however, that it would dominate the object structure or introduce a new kind of objects. As a language construct to specify autonomous activity, a simple keyword attached to operations is sufficient.

- Synchronization and mutual exclusion are operation-related. In restricting them to be not finer grained than that, they are object-oriented. They also integrate well into the inheritance structure. This way they can be employed to meet integrity-ensuring requirements as needed in an object class and also they avoid the problems of the "inheritance anomaly".

Moreover, the concept includes answers to problems arising with multiple inheritance in what the author considers to be the most natural way.

6. Further development

This paper has shown a concept of integrating two programming paradigms and has presented a collection of language constructs suitable to extend a given object-oriented language in a way consistent with our integration philosophy. The proposal does not include a detailed realization but leaves open various detail decisions. Especially our implementation concept for the rather complex synchronization mechanism was left out in order to focus on the conceptual aspects.

Moreover, progress in language design has not yet produced a general agreement on useful constructs for expressing concurrency even within the limits of any paradigmatic approach (e.g., explicit concurrency as in our proposal).

To take this situation into account and to keep our concept open to all variations that comply with its basic philosophy, we plan to pursue an open implementation by taking the essentials of the concept and incorporating them into a set of *intermediate language extension constructs*. The idea is that different high-level language approaches can then quickly be implemented by defining and realizing transformations from their concurrency constructs into our intermediate constructs.

Presentation of intermediate language details, including actual transformations implementing the features presented here, would be beyond the scope of this paper so as not to be confused with the higher-level concepts proposed and motivated here.

Practical considerations on realizing the our concept have been carried out in the context of the object-oriented language Eiffel ([Eiffel]). This work was originally planned to be a part of the Heron project ([Heron]) but then spread off as a separate work.

7. Conclusion

Our primary objection against too strong an adherence to the object structure when integrating further concepts stems from the consideration that objects are not in general a universal focus of programming. Object-orientation is a programming style equally valuable as others on an application-related basis. The dogmatic restriction to object-only languages (conceded, e.g., in [Meyer]) which has come into fashion, as well as its exaggerated amalgamation with, and simultaneous justification by software development methodologies, do not contribute to the unbiased progress in the development of modern languages and should not prevail in the languages of the future. Instead, good concepts like object-orientation, functional programming, data structures, a good concurrent programming paradigm still to be agreed on, etc., shall persist, and there will be the need to combine them in a compatible way, not to absorb one by the other.

On this conviction, our aim in integrating concurrency and object orientation is not to be forced into decisions between extreme points like "active" or "passive" objects but to follow a more liberal though still systematic way in the spirit of language design.

Moreover, in addition to considerations about concurrency in software construction, we have also identified *strong inherent reasons* against a closely related, completely unorthogonal integration model: It is the inheritance anomaly which gives us striking arguments against a close connection of the process and object concepts.

The separation of concerns applied in the solution of the paradigm combination conflicts in its different aspects is the primary exposal of the concept presented in this paper. There is no need, just because there are objects, to restrict the threading concept to them. On the other hand, there is no need, just because the thread concept is global and orthogonal, to either synchronize only on low-level or, as an emergency resort, to close objects against reasonable exploitment of concurrency.

We think the outlined approach would be an easier and more natural approach for both the programmers' views (when they want to implement some real-life modelling or some concurrent algorithm) and the implementor's task (not having the burden of frequent thread switches).

We also think the global thread model is an appropriate approach to the question of authorization of resource access by activities (concerning the *reflexive deadlock* problem).

As a suggestion how to implement such a concept without restricting its design space more than necessary, we have indicated the approach of intermediate constructs and transformation.

8. References

[Ada] Ken Shumate: *Understanding Concurrency in Ada.* McGraw-Hill 1988.

[Agha] G A Agha: Actors: *A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, MA, 1986.

[AWY] G Agha, P Wegner, A Yonezawa: *Research Directions in Object-Based Concurrency.* MIT Press 1993.

[BirNel] Andrew D Birrell, Bruce Jay Nelson: *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems 2, 1 (Feb 84), 39-59.

[Caromel] Denis Caromel: *A solution to the explicit/implicit control dilemma.* OOPS messenger 2, 2 (April 1991).

[Eiffel] Bertrand Meyer: *Eiffel: the language.* Prentice Hall 1992.

[EMERALD] A Black, N Hutchinson, E Jul, H Levy, L Carter: *Distribution and Abstract Types in Emerald.* IEEE transactions on software engineering SE-13, 1 (Jan 87).

[FRANCEZ] Nissim Francez: *Fairness.* Springer 1986.

[GOGUEN] Joseph Goguen: *On notation.* Keynote at TOOLS 10, Versailles 1993.

[HADDON] Bruce K Haddon: *Nested monitor calls.* ACM Operating Systems Review 11, 4 (Oct 77), 18-23.

[HERON] S Finke, P Jahn, O Langmack, K-P Löhr, I Piens, Th Wolff: *Distribution and Inheritance in the HERON Approach to Heterogeneous Computing.* Proc. 13. Int. Conf. on Distributed Computing Systems (ICDCS), Pittsburgh 1993.

[LISSCH] B Liskov, R Scheifler: *Guardians and Actions: Linguistic Support for Robust, Distributed Systems.* ACM transactions on programming languages and systems 5,3 (1982), 381-404.

[LÖHR] K-P Löhr: *Concurrency annotations.* Proc. 7. OOPSLA, Vancouver 1992.

[LOGLAN] Antoni Kreczmar, Andrzej Salwicki, Marek Warpechowski: *LOGLAN '88 – Report on the Programming Language.* LNCS 414, Springer 1990.

[MATSYONE] Satoshi Matsuoka, Akinori Yonezawa: *Analysis of inheritance anomaly in object-oriented concurrent programming languages.* In [AWY].

[MESEGUER] José Meseguer: *Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming.* Proc. ECOOP '93, LNCS 707.

[MEYER] Bertrand Meyer: *Eiffel: A Language and Environment for Software Engineering.* The Journal of Systems and Software 8 (1988), 199-246.

[NELSON] Michael L Nelson: *Concurrency & object-oriented programming.* ACM SIGPLAN Notices 26, 10 (Oct 1991).

[PAPATHOMAS] M Papathomas: *Concurrency Issues in Object-Oriented Programming Languages.* Object Oriented Development, Centre Universitaire d'Informatique, Université de Genève, 1989.

[OOCP] Akinori Yonezawa, Mario Tokoro (eds): *Object-Oriented Concurrent Programming.* MIT Press 1987.

[POOL] Pierre America: *POOL-T: A Parallel Object-Oriented Language.* In [OOCP].

[SR] Gregory R Andrews, Ronald A Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, Gregg Townsend: *An Overview of the SR Language and Implementation.* ACM ToPLaS 10, 1 (Jan 1988), 51-86.

[TRELLIS] J E B Moss, W H Kohler: *Concurrency Features for the Trellis/Owl Language.* BIGRE 54 (June 1987), 223-232.