Chapter 2

Concurrency in Object-Oriented Programming Languages

Michael Papathomas

Abstract An essential motivation behind concurrent object-oriented programming is to exploit the software reuse potential of object-oriented features in the development of concurrent systems. Early attempts to introduce concurrency to object-oriented languages uncovered interferences between object-oriented and concurrency features that limited the extent to which the benefits of object-oriented programming could be realized for developing concurrent systems. This has fostered considerable research into languages and approaches aiming at a graceful integration of object-oriented and concurrent programming. We will examine the issues underlying concurrent object-oriented programming, examine and compare how different approaches for language design address these issues. Although it is not our intention to make an exhaustive survey of concurrent object-oriented languages, we provide a broad coverage of the research in the area.

2.1 Introduction

Considerable research activity in the past few years has concentrated on the design of concurrent object-oriented programming languages (COOPLs). This research activity aimed at providing an integration of object-oriented and concurrent programming. The following points discuss some motivation for concurrent object-based programming:

- To augment the *modelling power* of the object-oriented programming paradigm. One goal of object-oriented programming can be seen as to model the real world directly and naturally [89]. Concurrency then adds to the modelling power by making it easier to model the inherently concurrent aspects of the real world.

Michael Papathomas, "Concurrency in Object-Oriented Programming Languages," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 31-68, Prentice Hall, 1995.

Reproduced with the permission of the Publisher, Prentice Hall (a Pearson Education company). This work is protected by copyright and may not be reproduced other than when downloaded and viewed on a single Central Processor Unit (CPU) for private use only. It is not otherwise to be reproduced or transmitted or made available on a network without prior written permission of Prentice Hall. All other rights reserved.

- To take advantage of the *software design benefits* of object-oriented programming and the potential for *software reuse* in the development of concurrent and distributed systems. Concurrent and distributed systems are becoming more widespread and the need to develop concurrent programs is becoming more common. This is witnessed by the support provided for concurrent programming at the application level provided by modern operating systems.
- To support *sharing of distributed persistent data*. The object-oriented paradigm lends itself well for providing location transparency by encapsulating within objects access to distributed persistent data. However, as information has to be shared, access to the objects has to be scheduled in a way that avoids interference and provides support for recovering from failures in the distributed environment. Although this could be left to the language implementation, as is the case in database management systems, taking advantage of the semantics of object types to ensure atomicity has substantial benefits with respect to performance and availability. This, however, requires the use of concurrency control mechanisms for the implementation of object types[90].
- To take advantage of *parallelism* in the implementation of object classes for increased execution speeds. Data abstraction can be used to conceal parallel implementations of objects from programs that use them so as to increase their performance when run on parallel machines. Parallelizing compilers could be used to generate parallel implementations of object classes, thus avoiding the need for concurrency constructs. However, better results are generally achieved by the use of explicit parallel algorithms as implicit approaches for parallel execution uncover and exploit only a number of restricted classes of parallelism [46]. Moreover, as data abstraction hides the details of the implementation of classes, users of these classes need not be aware of their concurrent implementation.

In all of the above cases it is necessary to combine the concurrent and object-oriented programming paradigms, provide linguistic support for concurrent object-oriented programming and, ideally, exploit the reuse potential of object-oriented programming for concurrent software.

However, combining object-oriented and concurrency features has proven to be more difficult than might seem at first sight. Clearly, devising a language that has both concurrent programming and object-oriented constructs poses no problem. There has been a large number of proposals for combining object-oriented and concurrency features. However, they are not equally successful in drawing the benefits of object-oriented programming for concurrent software development. The problem is that these features are not orthogonal, and consequently they cannot be combined in an arbitrary way. Most of the research in the area is devoted to devising graceful combinations that limit the interference of features.

In this chapter we present a design space for the approaches for combining object-oriented and concurrency features and a set of criteria for evaluating the various choices. We use the criteria to evaluate some proposals and identify approaches that do not

adequately support object-oriented programming as well as approaches that do achieve a graceful combination of the features.

In section 2.2 we present a design space for combining object-oriented and concurrency features with respect to several aspects of language design. In section 2.3, we discuss the issues that have to be addressed to provide the benefits of object-oriented programming. Section 2.4 examines the impact of some proposals on the integration of the programming paradigms and their potential for reuse. Finally, in section 2.5 we present our conclusions, discuss open problems and directions for further work in the area.

2.2 Design Space

We start by presenting three aspects of COOPLs that we consider for constructing the design space, and then we discuss the design choices with respect to each of these aspects. Later, in section 2.4, we will examine more closely some existing languages showing how the design of their features situate them in the design space.

2.2.1 A Design Space for Concurrent Object-Oriented Languages

We seek to evaluate language design choices with respect to the integration of their concurrency and object-oriented features and the degree to which software reuse is supported. In particular, we wish to understand how choices of concurrency constructs interact with object-oriented techniques and affect the reusability of objects. As such, our classification scheme concentrates on the relationship between objects and concurrency. We shall consider the following aspects:

- *Object models:* how is object consistency maintained in the presence of concurrency? The way objects are considered with respect to concurrent execution may or may not provide them with a default protection with respect to concurrent invocations. Furthermore, different languages may favour or enforce a particular way of structuring programs to protect objects.
- *Internal concurrency:* can objects manage multiple internal threads? This issue concerns the expressive power that is provided to objects for handling requests. Note that the execution of internal threads is also related to the protection of the internal state objects, which is determined by the choice of object model.
- *Constructs for object interaction:* how much freedom and control do objects have in the way that requests and replies are sent and received? The choice of concurrency constructs for sending and receiving messages determines the expressive power that is provided for implementing concurrent objects. Moreover, the design of constructs for conditional acceptance of messages interacts with the use of class inheritance.

In the presentation of the design space, it will become apparent that these aspects are not entirely independent: certain combinations of choices are contradictory and others are redundant or lack expressive power.

2.2.2 Concurrent Object Models

There are different ways one can structure a concurrent object-based system in order to protect objects from concurrency. A language may support constructs that favour or even enforce one particular way, or may leave it entirely to the programmer to adopt a particular model. There are three main approaches:

- *The orthogonal approach:* Concurrent execution is independent of objects. Synchronization constructs such as semaphores in Smalltalk-80 [40], "lock blocks" as in Trellis/Owl [68] or monitors as in Emerald [19] must be judiciously used for synchronizing concurrent invocations of object methods. In the absence of explicit synchronization, objects are subject to the activation of concurrent requests and their internal consistency may be violated.
- *The homogeneous approach:* All objects are considered to be "active" entities that have control over concurrent invocations. The receipt of request messages is delayed until the object is ready to service the request. There is a variety of constructs that can be used by an object to indicate which message it is willing to accept next. In POOL-T [6] this is specified by executing an explicit accept statement. In Rosette [83] an *enabled set* is used for specifying which set of messages the object is willing to accept next.
- *The heterogeneous approach:* Both active and passive objects are provided. Passive objects do not synchronize concurrent requests. Examples of such languages are Eiffel // [26] [27] and ACT++ [45]. Both languages ensure that passive objects cannot be invoked concurrently by requiring that they be used only locally within single-threaded active objects. Argus [55] provides both *guardians* (active objects) and CLU *clusters* (passive objects) [52].

2.2.3 Internal Concurrency

Wegner [87] classifies concurrent object-based languages according to whether objects are internally sequential, quasi-concurrent or concurrent:

- *Sequential objects* possess a single active thread of control. Objects in ABCL/1 [94] and POOL-T and Ada tasks [9] are examples of sequential objects.
- *Quasi-concurrent objects* have multiple threads but only one thread may be active at a time. Control must be explicitly released to allow interleaving of threads. Hybrid domains [47][70][71][72] and monitors [42] are examples of such objects.

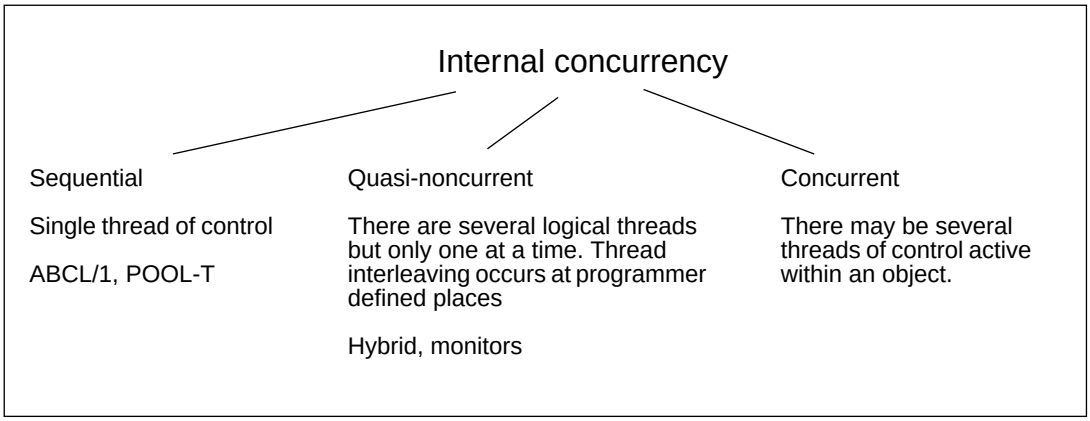

Figure 2.1 *Approaches to internal concurrency.*

- *Concurrent objects* do not restrict the number of internal threads. New threads are created freely when accepting requests. Ada *packages* and POOL-T *units* resemble concurrent objects (though they are not first-class objects). Languages like Smalltalk-80 that adopt the orthogonal object model also support concurrent objects. From the point of view of the called objects, a new local thread is effectively created whenever a method is activated in response to a message.

According to the above classification, the threads of concurrent objects are created freely when an object receives a message. However, there are languages where objects may have internally concurrent threads that are not freely created by message reception. In order to include these languages in the classification and to capture more information about the way that threads are created, we generalize the concurrent object category to include any language in which objects have concurrent threads, irrespective of the way they are created, and consider separately the issue of thread creation.

We identify three, non-exclusive ways for the creation of threads within objects as follows:

- *By message reception*: Thread creation is triggered by reception of a message. An object cannot create a thread on its own unless it can arrange for a message to be sent to it without blocking the currently executing thread. Depending on whether objects may control the creation of threads, we have the following subcategories:
  - — *Controlled by the object:* The object may delay the creation of threads. For example, in the language Sina [84] a new concurrent thread may be created for the execution of a method belonging to a select subset of the object's methods only if the currently active thread executes the *detach* primitive.
  - — *Unconstrained creation:* Threads are created automatically at message reception. This is the default for languages with an orthogonal object model.
- *Explicit creation*: Thread creation is not triggered by message reception but the object itself initiates the creation of the new thread. For instance, in SR [12] there is a construct similar to a "cobegin" [11] to initiate the execution of concurrent threads.

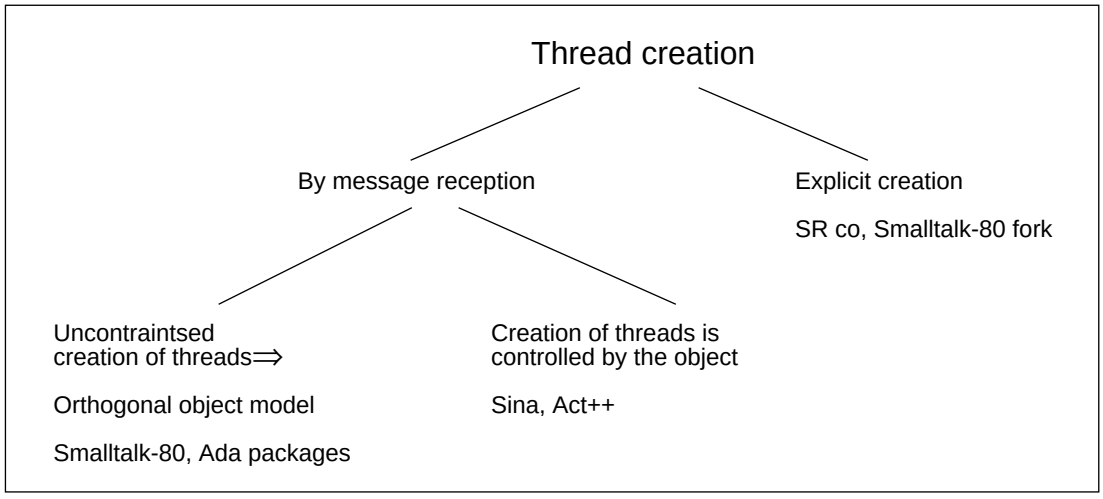

Figure 2.2 *Approaches to thread creation.*

Another way to create a new thread, in the absence of a special construct, is to call asynchronously an operation of the object. This requires, however, that such calls are not blocked at the object's interface. This approach is used in a recent version of Sina. Such calls bypass the normal method synchronization constraints as well as the request queue at the object's interface. Finally, it would also be possible to create new independent objects to call the object methods in parallel. However, this is cumbersome and it also requires some means of bypassing the message queue at the object's interface.

The *next* and *become* primitives in Rosette and ACT++ can be viewed as a controlled creation of threads, with the additional restriction that concurrent threads may not share the object's state since they execute on different "versions" of the object.

In Guide [48], an object may be associated with a set of activation conditions that specify which methods may be executed in parallel by internally concurrent threads. In the default case, as with any language following an orthogonal approach for concurrency, objects may be viewed as concurrent with unconstrained creation of threads triggered by external messages.

The creation of threads by reception of external messages or by execution of a special construct are neither mutually exclusive design choices — as illustrated by SR, which supports both — nor redundant, as we will see in section 2.3.

2.2.4 Constructs for Object Interaction

We classify these constructs with respect to the degree of control that can be exercised by objects in the client and server roles. We specifically consider *reply scheduling*, which concerns the degree of flexibility the client has in accepting a reply, and *request scheduling*, which concerns the control the server can exercise in accepting a request.

2.2.4.1 Issuing Requests

The following important issues can be identified with respect to the constructs supported for issuing requests:

- *Addressing:* How are the recipients of a request specified and determined? How and where is the reply to be sent? Flexible control over the reply destination can reduce the amount of message passing required.
- *Synchronization for requests and replies:* Can the current thread continue after issuing the request? What mechanisms are supported for matching replies to requests? How does the client synchronize itself with the computation and delivery of the reply?
- *First-class representation of requests and replies:* Do requests and replies have a first-class representation that permits them to be forged or changed dynamically? What aspects (e.g. destination, method name) can be changed dynamically?

We further discuss these issues below and present how they are addressed by different proposals.

Addressing

In most languages the recipient of a request is specified directly by using its object identifier. However, there are some proposals allowing for more flexible ways of addressing where the system determines the recipient of the request. We review some of these proposal below.

Types as Recipients in PROCOL

In PROCOL [49] [85] an object type may be used to specify the recipient of a request. In this case the potential recipients are any instance of the type that is in a state such that it may accept the request. The system determines one recipient among the set of potential recipients and delivers the request. It is important to note that this feature does not support any form of multicast; exactly one message is exchanged with the chosen recipient in a point to point fashion.

ActorSpace

ActorSpace [2] is a general model providing a flexible and open-ended approach to object communication that has been developed in the context of the actor model.

In this mode, *destination patterns* may by used to designate the recipients of a request. Patterns are matched against attributes of actors in an specified actorspace — a passive container of actors — to determine a set of potential recipients. A message may be sent by either one of two primitives: *send* or *broadcast*.The former delivers exactly one message to a recipient chosen non-deterministic by the system. The latter provides a form of multicast by delivering the request to all potential recipients.

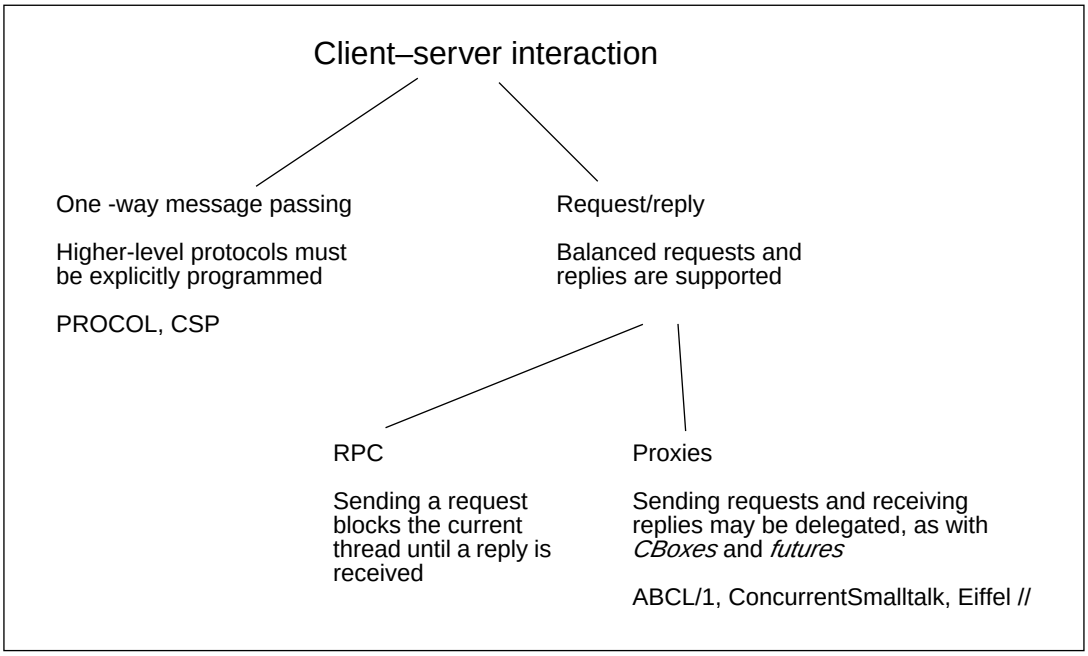

Figure 2.3 *Client–server interaction mechanisms.*

Extra flexibility is provided in this model by allowing the dynamic inclusion and removal of actors from ActorSpaces as well as by allowing the dynamic modification of actor attributes. Moreover, ActorSpaces may be nested.

Synchronization for Requests and Replies

We initially distinguish between *one-way message passing* communication primitives and constructs supporting a *request/reply* protocol. The latter provide support for object interactions where requests will be eventually matched by replies. These mechanisms vary in flexibility when sending requests and receiving replies. Strict RPC approaches enforce that requests will be matched by a reply and delay the calling thread until the reply is available. Further flexibility is provided by "proxy" objects which disassociate the sending or receiving of messages from the current thread of control. Examples of built-in proxy objects are *future variables* [94] and *CBoxes* [92].

One-Way Message Passing

Whether communication is synchronous with one-way message passing, as in CSP [43] or PROCOL [85], or asynchronous, as in actor languages, clients are free to interleave activities while there are pending requests. Similarly, replies can be directed to arbitrary addresses since the delivery of replies must be explicitly programmed.

The main difficulty with one-way message passing is getting the replies. The client and the server must cooperate to match replies to requests. As we shall see in section 2.3, the additional flexibility and control provided by one-way message passing over request/reply based approaches can only be properly exploited if objects (i.e. servers) are implemented in such a way that the reply destination can always be explicitly specified in a request.

Remote Procedure Call

With RPC the calling thread of the client is blocked until the server accepts the request, performs the requested service and returns a reply. Most object-oriented languages support this form of interaction, though "message passing" is generally compiled into procedure calls.

Supporting RPC as the only means for object interaction may be a disadvantage when objects are sequential as we will see in the next section. Although it is trivial to obtain a reply, it is not possible to interleave activities or to specify reply addresses.

Proxies

An alternative approach that provides the client with more flexibility in sending and receiving replies is to introduce *proxies*. The main idea is to delegate the responsibility of delivering the request and obtaining the reply to a proxy. (The proxy need not be a first-class object, as is the case with *future variables* [94].) The actual client is therefore free to switch its attention to another activity while the proxy waits for the reply. The proxy itself may also perform additional computation or even call multiple servers.

If necessary, the reply is obtained by the original client by an ordinary (blocking) request. This approach, variants of which are supported by several languages [27][94][92], maintains the benefits of an RPC interface and the flexibility of one-way message passing. In contrast to one-way message passing, however, there is no difficulty in matching replies to requests.

A closely related approach is to combine RPC with one-way message passing. In ABCL/1, for example, an object that externally has an RPC interface may internally use lower-level message-passing primitives to reply by sending an asynchronous message to the client or to its proxy. The use of such facilities is further discussed in section 2.4.2.

First-Class Representation of Requests and Replies

The ability to have a first-class representation of requests and replies may enhance substantially the expressive power of a language. There is a range of aspects of requests and replies that may have a first-class representation in a language. This varies from (almost) no first-class representation at all to a full first-class representation of all aspects of requests and replies. Below we discuss how this issue is addressed in some languages that are characteristic of the various possibilities.

Minimal First-Class Representation

Apart from the method's arguments and the target, all other aspects, such as the method name and the return address, cannot be specified dynamically. This the case for languages such as POOL-T, Hybrid and Trellis/Owl. One could argue that since the target and the arguments can be specified at run-time, there is a first-class representation of some aspects and that the categorization is not accurate. In fact, in older language proposals such as CSP [43] the targets of messages were determined statically. This, however, is uncommon in more recent languages since it makes it hard to develop software libraries: a server that must be statically bound to its potential callers has a low reuse potential. A first-class representation of the target and arguments can be considered as a minimum that one should expect to find in every language.

First-Class Representation of Method Names and Reply Addresses

PROCOL supports the first-class representation of method names. The name of the method to call may be supplied as a string. This allows the method names for a request to be passed in messages or computed at run-time.

With ABCL/1 it is possible to specify dynamically and explicitly the object that is to receive the reply of a request. The benefits of the use of this feature are discussed in section 2.4.2.

Full First-Class Representation

As one would expect, full first-class representation of requests is provided in reflective languages such as ABCL/R. However, it is also provided in languages such as Smalltalk and Sina which are not fully reflective. In fact, the latter two illustrate the usefulness and the possibility of having such features in any concurrent language which is not fully reflective. Briot [23] has used the features of Smalltalk to build a several object-oriented programming models using the relative primitive concurrency features provided in the Smalltalk system. Aksit *et al.* [4] show how these features may be used to abstract and reuse several object coordination paradigms.

2.2.4.2 Accepting Requests

A main concern from the point of view of an object acting as a server is whether requests can be conditionally accepted.* When a request arrives, the server may be busy servicing a previous request, waiting itself for a reply to request it has issued, or idle, but in a state that requires certain requests to be delayed. We distinguish initially between conditional and unconditional acceptance of requests. Conditional acceptance can be further discriminated according to whether requests are scheduled by explicit acceptance, by activation conditions or by means of reflective computation (see figure 2.4).

* A secondary issue is whether further activity related to a request may continue after the reply has been sent as in the Send/Receive/Reply model [39], but this can also be seen as concern of internal concurrency where follow-up activity is viewed as belonging to a new thread.

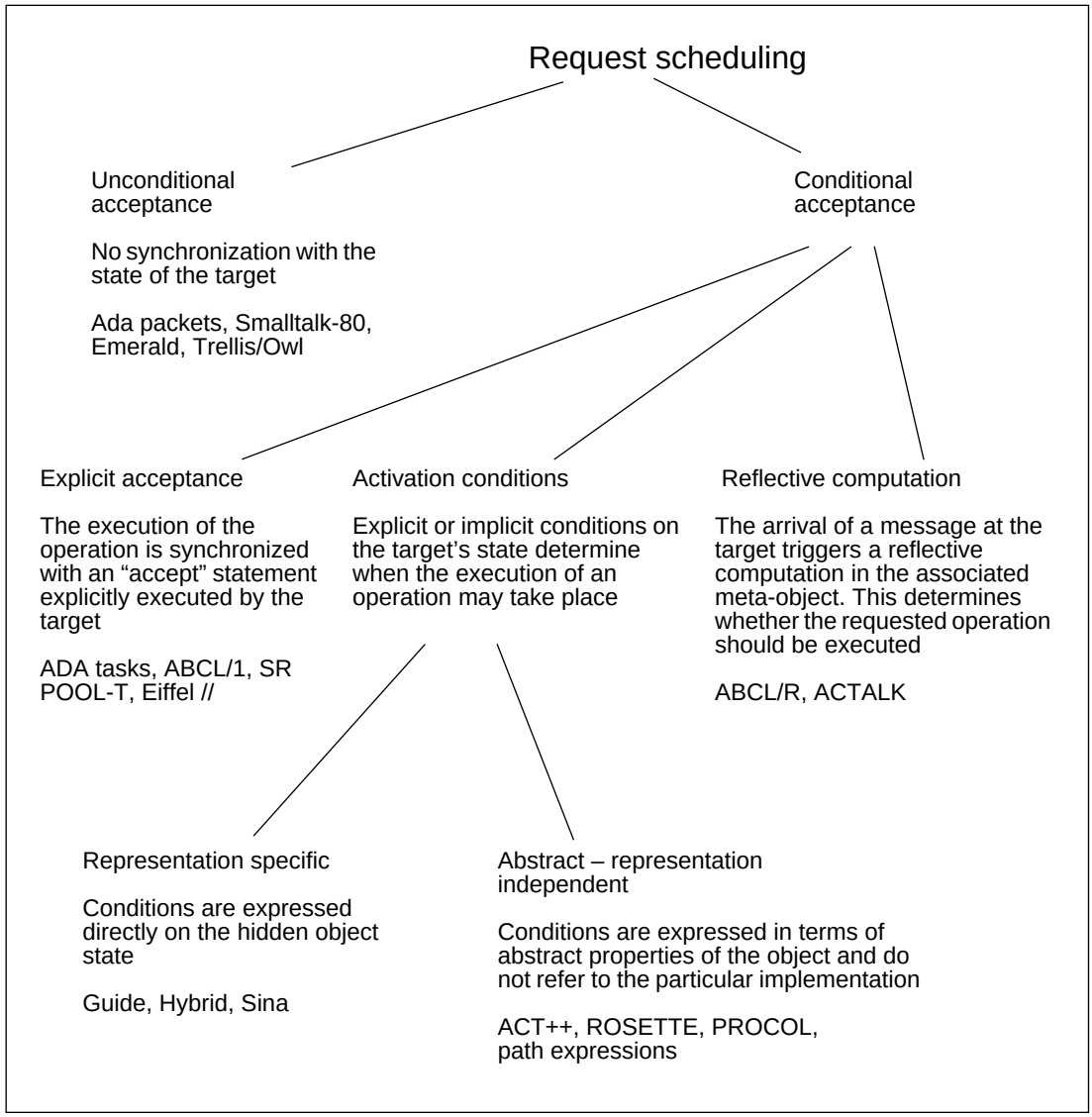

Figure 2.4 *Approaches to scheduling requests.*

Unconditional Acceptance

Unconditional acceptance of requests is illustrated by monitors [42] and by Smalltalk-80 [40] objects. The mutual exclusion that is provided by monitors could be considered as an implicit condition for the acceptance of requests. However, the mutual exclusion property is captured by viewing monitors as quasi-concurrent objects so we consider request acceptance to be unconditional. Note that message acceptance for languages with an orthogonal object model is by default unconditional.

Explicit Acceptance

With *explicit acceptance*, requests are scheduled by means of an explicit "accept" statement executed in the body of the server. Accept statements vary in their power to specify which messages to accept next. Acceptance may be based on message contents (i.e. operation name and arguments) as well as the object's state. Languages that use this approach are Ada, ABCL/1, Concurrent C, Eiffel//, POOL-T and SR. With this approach objects are typically single-threaded, though SR is an exception to this rule.

Activation Conditions

With *activation conditions*, requests are accepted on the basis of a predicate over the object's state and, possibly, the message contents. The activation condition may be partly implicit, such as the precondition that there be no other threads currently active within the object. An important issue is whether the conditions are expressed directly over a particular representation of the object's state or if they are expressed in more abstract terms. In Guide, for example, each method is associated with a condition that directly references the object's instance variables, whereas in ACT++ the condition for accepting a message is that the object be in an appropriate *abstract state* which abstracts from the state of a particular implementation. Another approach is to specify the legal sequences of message acceptance by means of a regular expression, as in path expressions [24] and PROCOL [85].

There are also some proposals such as *synchronizers* [38], *separate method arguments* [66] and *state predicates* [74], for activation conditions that depend on the state or the computation history of other objects.

A synchronizer [38] is a special object associated with a group of objects. When a method of any of these objects is called a condition in the synchronizer is evaluated. Depending on the outcome, the execution of the method may proceed, or be delayed until the condition becomes true. Synchronizers may have their own variables that are used to store information about the computation history of a group of objects.

Separate method arguments [66] can be used to constraint the execution of a method by preconditions on the argument declared as "separate." The execution of the method is delayed until the preconditions are true and the separate objects are "reserved" for the duration of the call. That is, they can only be used in the body of a method.

With state predicate notifiers [74], the execution of a method can be constrained by the notification that another object has reached a state that satisfies a *state predicate*.This feature has synchronous and asynchronous forms. In the synchronous variant, the notifying object waits until the method is executed and the method gains exclusive access to the object. In the asynchronous variant the notifying object proceeds independently.

Reflective Computation

With *reflective computation* the arrival of a request triggers a method of the server's *meta-object*. The meta-object directly then manipulates object-level messages and mailboxes as

objects. This approach is followed by the language ABCL/R [86] and it is also illustrated in Actalk [23] where some reflective facilities of the Smalltalk-80 system are used to intercept messages sent to an object and synchronize their execution in a way that simulates message execution in actor-based languages.

2.3 Criteria for Evaluating Language Design Choices

So far we have presented a design space covering the most significant choices in the design of concurrency features for OOPLs, but we have said little about how the various approaches compare. Since our goal is to arrive at COOPLs that provide the advantages of object-oriented programming for the development of concurrent systems, we must first formulate our requirements as precisely as possible, before beginning to compare the approaches. We first discuss the issue of developing object classes that have high reuse potential. Then, we turn our attention to the support for reuse at a finer granularity than objects and examine the issues related to the use of inheritance and the reuse of synchronization constraints.

2.3.1 Object-Based Features — Support for Active Objects

The main issue for reuse at the object level is that concurrency in an object-oriented language should not diminish the benefits of object-based features with respect to reuse. For instance, encapsulation should still protect the internal state of objects from surrounding objects and it should still be possible to insulate objects' clients from implementation choices. This should make it possible to change the implementations without affecting the clients provided that the interfaces are maintained and that changes are, in some sense, behaviourally compatible.

Object-oriented and concurrent programming have different aims that incur different software structuring paradigms. Object-oriented programming aims at the decomposition of software into self-contained objects to achieve higher software quality and to promote reusability. Concurrent programming aims at expressing and controlling the execution, synchronization and communication of conceptually parallel activities. Its primary goal is to provide notations that are suitable for devising solutions to problems that involve the coordination of concurrent activities [11].

In order to compare language designs it is necessary to adopt a programming model for concurrent object-based programming and evaluate how well the various languages support this model. Our view regarding the way the two programming paradigms should be combined is by structuring programs as cooperating objects that exchange messages. This is similar to the way sequential object-oriented programs are structured, however, in concurrent programs objects may encapsulate one or more concurrent threads that implement their behaviour. Moreover, the operations of an object may be invoked by concurrently executing objects.

We use the term *active objects* for this programming model to emphasize that objects themselves rather than the threads that invoke their operations have the responsibility to schedule concurrent requests. Requests should be scheduled in a way consistent with the object's internal state and the possibly spontaneous execution of internal threads. The objects developed following this model are independent self-contained entities. They can be reused across applications and they may be refined to support different scheduling policies for invoked operations. The programs that use the objects should not be affected by such changes.

Although any language combining concurrent and object-oriented features could be used to develop software following this model, as will be illustrated in section 2.4, not all combinations of concurrent and object-oriented features are equally successful in supporting this programming model. Below we develop a number of requirements on the language features to adequately support programming following an active object model. In section 2.4 we will use these requirements to evaluate language design choices and identify the shortcomings of some approaches.

2.3.1.1 Requirements

According to the active object model discussed above, we would like languages to support the development of self-contained objects with high reuse potential. A general principle for achieving this is that reusable object classes should make minimal assumptions about the behaviour of applications that will use them. Furthermore, the choice of constructs should not constrain the possible implementations of a class. We can formulate our requirements as follows:

1. *Mutual exclusion — protecting the objects' state:* The internal state of objects should be automatically protected from concurrent invocations so that it will be possible to reuse existing objects in concurrent applications without modification.
2. *Request scheduling transparency:* An object should be able to delay the servicing of requests based on its current state and on the nature of the request. This should be accomplished in a way that is transparent to the client. Solutions that require the cooperation of the client are not acceptable from the point of view of reusability since the client then cannot be written in a generic fashion.
3. *Internal concurrency:* The concurrency constructs should allow for the implementation of objects that service several requests in parallel or that make use of parallelism in their implementation for increased execution speed in the processing of a single request. This could be done either by supporting concurrent threads within an object or by implementing an object as a collection of concurrently executing objects. Whatever approach is chosen, it is important that internal concurrency be transparent to the object's clients so that sequential implementations of objects may be replaced by parallel ones.
4. *Reply scheduling transparency:* A client should not be forced to wait until the serving object replies. In the meantime it may itself accept further requests or call other objects in parallel. It may even want replies to be directly sent to a proxy. Request

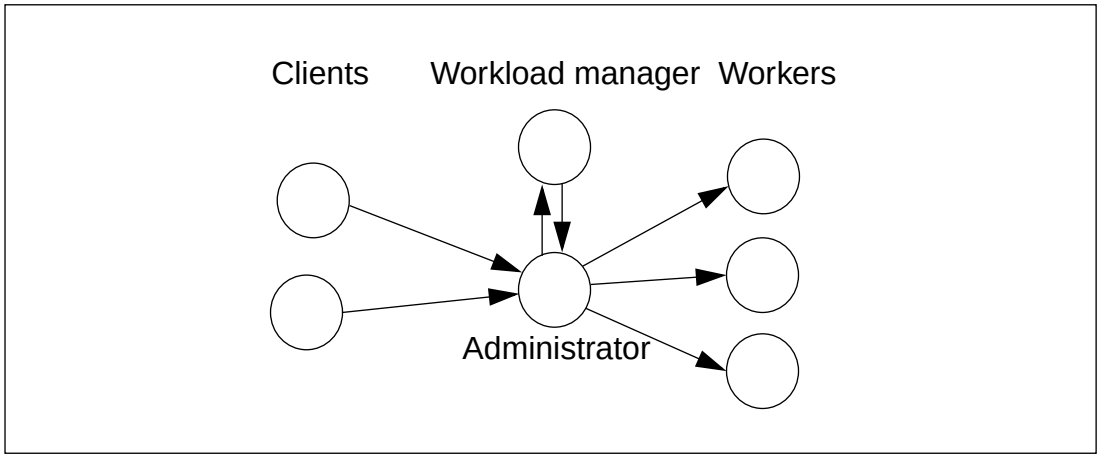

Figure 2.5 *The administrator example.*

scheduling by the client should not require the cooperation of the server since this would limit the ability to combine independently developed clients and servers.

2.3.1.2 An Example

In order to compare the design choices and their combinations with respect to the reuse requirements, we shall refer to an instance of a "generic" concurrent program structure: the *administrator* inspired by [39]. The administrator is an object that uses a collection of "worker" objects to service requests. An administrator application consists of four main kinds of components. The *clients* issue requests to the administrator and get back results. The *administrator* accepts requests from multiple concurrent clients and decomposes them into a number of subrequests. The *workload manager* maintains the status of workers and pending requests. *Workers* handle the subrequests and reply to the administrator. The administrator collects the intermediate replies and computes the final results to be returned to clients (see figure 2.5).

The administrator is a very general framework for structuring concurrent applications. For example, workers may be very specialized resources or they may be general-purpose compute servers. The workload manager may seek to maximize parallelism by load balancing or it may allocate jobs to workers based on their individual capabilities.

The components described above identify functionally distinct parts of the application that could have been developed independently and reused as indicated above to construct a new application.These components do not have to be implemented as single objects, and indeed, as we see later, depending on the constructs provided by certain languages, several objects will be necessary for realizing the desired functionality. However, it should be possible to modify the implementation of the above components without affecting the rest as if they were single objects.

The following points relate the language design requirements listed above to the reuse issues in the case of the example application:

- *Mutual exclusion:* (i) Workload manager reuse – the workload manager must be protected from concurrent requests by the administrator. There may be cases where the administrator does not invoke the workload manager concurrently. Although in such cases no protection is needed, workload managers that are not protected could not be reused in different concurrent implementations of the administrator. In such a concurrent implementation the administrator may use a collection of proxies that may invoke the workload manager concurrently. (ii) Worker reuse – workers should similarly be protected so that arbitrary objects may be used as workers with various implementations of the administrator, including concurrent ones.
- *Request scheduling transparency:* (iii) Genericity of clients, reusing the administrator with different clients — the administrator must be able to interleave (or delay) multiple client requests, but the client should not be required to take special action. In fact it should be possible to implement any object as an administrator and it should not matter to the object's clients if the serving object happens to be implemented as an administrator.
- *Internal concurrency:* (iv) Client/worker reuse — the administrator should be open to concurrent implementation (possibly using proxies) without constraining the interface of either clients or workers;
- *Reply scheduling transparency:* (v) Worker reuse — it must be possible for the administrator to issue requests to workers concurrently and to receive their replies when it chooses without special action by workers;

2.3.2 Inheritance and Synchronization

There are two main issues concerning reuse at a finer granularity than objects.

- The first is to maintain in concurrent languages the reuse potential offered by inheritance in sequential languages. Several early papers have reported difficulties in using class inheritance in COOPLs as well as in the design of languages that integrate class inheritance and concurrency constructs [19] [6] [22]. In some cases inheritance was left out as it was deemed difficult to integrate and of limited use. The need to synchronize the execution of inherited, overridden and newly defined methods, without breaking the encapsulation between classes, makes it more difficult to take advantage of class inheritance than in sequential languages. For instance, if mutexes are used for synchronizing method execution, a method defined in a subclass would have to access a mutex defined in a superclass in order to be synchronized with superclass methods. This would break encapsulation between classes. The design of concurrency constructs should be made in way to avoid such problems.
- The second is to make it possible to reuse algorithms, often called *synchronization constraints*, for scheduling the execution of methods of a class. For instance, a class may implement a synchronization algorithm that schedules its methods according to the readers and writers synchronization scheme. It would be desirable to be able to

reuse this algorithm in other classes taking into account the reader/writer property of its methods.

In most languages the reuse of synchronization constraints is achieved through class inheritance and the term *inheritance of synchronization constraints* is often used for this issue. We have chosen the term *reuse of synchronization constraints* since class inheritance is only one possible means to achieve reuse. Furthermore, it is questionable whether class inheritance should be used for this purpose. We will further elaborate on this point below. Then, we will discuss separately the requirements for supporting class inheritance and for reusing synchronization constraints.

Inheritance is often considered as the most prominent feature of object-oriented programming. The most widespread object-oriented languages such as C++, Smalltalk and Eiffel provide an inheritance mechanism that may be used for different purposes. These include: the reuse of the implementation of a class in the implementation of a new class; the specification of a type compatibility relation between a class and its parent classes, considering for type-checking purposes that instances of the class are of the same type as instances of its superclasses; finally, it may be used to express that the concept or entity modelled by the subclass is, in some sense, a refinement of the concepts or entities represented by its parent classes.

The use of a single mechanism for all these purposes can, on one hand, be a source of confusion and on the other, limit the effectiveness of the mechanism for each of these different purposes. For instance, subtypes have to be related to a class inheritance relationship even if they do not share any part of their implementation. In order to use part of the implementation of a class in a new class, all the methods have to be inherited to comply with the subtype relation that is also expressed by the inheritance link.Wegner and Zdonik [88] provide a general and in-depth discussion of inheritance as an incremental modification mechanism and illustrate its use for different purposes. Guide [48] and POOL-I [8] are concrete examples of languages with mechanisms that distinguish between the different uses of inheritance. Both languages distinguish between class inheritance as a code reuse mechanism and typing. POOL-I goes even further by also allowing the specification of behaviourally compatible classes.

In section 2.4.3 we will examine more closely the approaches for the reuse of synchronization constraints followed by different languages. This will illustrate the interactions that class inheritance may have with the reuse of synchronization constraints in these different approaches.

2.3.2.1 Class Inheritance

The issues listed below have to be addressed in order to take advantage effectively of the reuse potential of inheritance. The first two are concerned with the reuse of superclass methods. The third one concerns the use of inheritance for providing generic algorithms through the definition and refinement of *abstract classes* [36] [44].

- *Separate specification of the synchronization constraints:* If the code that implements the synchronization decisions related to the execution of methods is included

directly in methods, inherited methods typically will have to be modified to account for the synchronization constraints of the subclass [45].

- *Interface between methods and the synchronization constraints:* The separate specification of synchronization control actions and method code does not necessarily mean that the execution of methods once started should be carried out without any further interaction with the synchronization constraints. Such an approach limits the expressive power of a language. Instead, there should be a well-defined interface between methods and the synchronization constraints that allows several actions in the execution of the method to interact with the synchronization constraints associated with the various classes where it is reused.
- *Consistency with other uses of inheritance for software composition:* Apart from reusing individual methods, inheritance serves to facilitate sharing of algorithms and designs [36]. For this purpose, inheritance is paired with other features such as invocation of methods through pseudo-variables such as *self* or *super* in Smalltalk.

2.3.2.2 Reuse of Synchronization Constraints

The issues discussed below are important for evaluating and comparing the proposals for the specification and reuse of synchronization constraints:

- *Flexibility of the binding mechanism:* The mechanism that is used to apply constraints to a particular class determines the flexibility with which constraints may be reused. Depending on the mechanism, constraints are bound to exactly one class (the class where they were introduced), or to any class that inherits from the class that introduced the constraints. Additionally, method names appearing in a constraint specification may be considered as variables to be substituted at binding time with method names defined in a particular class.
- *Compositionality and extensibility:* This concerns the support provided for reusing previously defined constraints in the definition of new ones. A related issue is extending the application of constraints to methods that are introduced at a later stage.
- *Polymorphism:* The potential applicability of constraints to different classes. This is related to the binding mechanism and modularity; constraints could be specified in a way that would allow them to be applied to different classes. However, this may be impossible or inconvenient because of the absence of an appropriate binding mechanism.
- *Modifiability and locality of change:* There are circumstances where it may be desirable or necessary to change the implementation of a class or of just the synchronization constraint. Depending on the approach, this may be achieved easily through some local modification or it may require a cascade of changes in synchronization constraints. In some cases it may even be needed to modify the inheritance hierarchy. Most of the other aspects discussed above come into play when considering this issue.

2.4 Exploring the Language Design Space

We now propose to compare the various approaches to the design of COOPLs by systematically exploring the language design space and evaluating design choices against the requirements specified in the previous section. Since the various aspects of the design space are sometimes intertwined, we will find ourselves returning to common issues on occasion. Basically we will take the following course: first we briefly consider the three categories of object models; then we consider object interaction mechanisms in combination with internal concurrency; finally we explore inheritance and synchronization constraints as a topic worthy of separate study. We summarize our conclusions in section 2.4.4.

2.4.1 Object Models

By the requirement of mutual exclusion, we can immediately discount the orthogonal object model as it provides no default protection for objects in the presence of concurrent requests. The reusability of workers and workload managers is clearly enhanced if they will function correctly independently of assumptions of sequential access.

The heterogeneous model is similarly defective since one must explicitly distinguish between active and passive objects. A generic administrator would be less reusable if it would have to distinguish between active and passive workers. Similarly worker reusability is weakened if we can have different kinds of workers.

The *homogeneous* object model is the most reasonable choice with respect to reusability. No distinction is made between active and passive objects.

Note that it is not clear whether the performance gains one might expect of a heterogeneous model are realizable since they depend on the programmer's (static) assignment of objects to active or passive classes. With a homogeneous approach, the compiler could conceivably make such decisions based on local consideration — whether a component is shared by other concurrently executing objects is application specific and should be independent of the object type.

2.4.2 Object Interaction Mechanisms

Request-reply mechanisms such as an RPC-like interface provide more support for object reuse. Using our administrator example, we can see that one-way message passing has several disadvantages over RPC for reusing objects.

A concurrent client may issue several requests to the administrator before it gets a reply. In this case it is important for the client to know which reply corresponds to which request. Are replies returned in the same order as requests? In the case of synchronous message passing an additional difficulty is that the administrator may get blocked when it sends the reply until the client is willing to accept it. Requiring the client to accept the reply imposes

additional requirements on the client and makes reuse more difficult. Either a different mechanism has to be supported for sending replies or proxies have to be created.

One-way message passing is also inconvenient for coping with the interaction between the administrator and worker objects. A difficulty with using one-way messages is getting the replies from workers: as there will be several workers that are invoked in parallel, as well as potentially concurrent invocations of single worker, it can be difficult for the administrator to tell which reply is associated with which request.

A solution to this problem is to create a proxy for each request. The proxy would carry out the request and then send a message to the administrator containing the worker's reply plus some extra information used for identifying the request. As with sequential RPC the administrator will also have to manage local queues for partially completed requests.

2.4.2.1 Sequential Objects

We argued that an RPC interface for objects provides better support for object reuse than one-way message passing. However, we quickly discover that if objects have a single thread of control and RPC is the only communication mechanism, the request and reply scheduling requirements of the administrator are not satisfied. We further discuss the limitation of this design choice combination below. Then we show additional mechanisms that may be used to overcome these limitations without giving up the RPC-interface or completely discarding sequential object design choice. The limitation of the combination of sequential objects ("modules" in their case) and RPC is discussed at length in [54]. However, they reach the conclusion that either the sequential object or the RPC choice should be discarded.

Limitations of the Sequential Object–RPC Combination

In particular, a sequential RPC administrator will not be able to interleave multiple clients' requests as it will be forced to reply to a client before it can accept another request. The only "solution" under this assumption requires the cooperation of the client, for example: the administrator returns the name of a "request handler" proxy to the client, which the client must call to obtain the result. In this way the administrator is immediately free to accept new requests after returning the name of the request handler. Such an approach is, however, incompatible with the requirement on request scheduling transparency since scheduling of requests by the administrator is not transparent to its clients.

Consider for instance that we would like to replace the sequential implementation of an existing object class by a parallel implementation where instances of the class act as administrators for a collection of appropriate worker objects. In accord with our requirements we would like to take advantage of encapsulation and data abstraction to replace the old implementation without having to modify the programs that used it. This, however, is not possible since, as discussed above, in order to be able to process client requests concurrently, an object, implemented as an administrator, has to have a different interface than an object having a sequential implementation.

The sequential RPC combination also provides limited support for reply scheduling by the administrator. If the administrator invokes workers directly using RPC, its single thread will get blocked until the invoked worker computes the result and returns the reply. The sequential RPC combination prevents the administrator from invoking several workers in parallel, or accepting further client requests while a worker computes the result and receiving the workers' replies at a later time.

It is also possible to have the workers cooperate with the administrator so that it does not block when delegating work to them, but such solutions require workers to be coded in a special way to implement the cooperation. This is incompatible with our requirement of request scheduling transparency, which would allow any object to be potentially used as a worker.

Using Proxies for Reply Scheduling

The limitation of the sequential RPC combination for reply scheduling can be overcome by the use of "courier" proxies used by the administrator to invoke workers. Each time the administrator needs to invoke a worker it creates an appropriate courier proxy that will invoke the worker instead. To get a worker's reply, the administrator could invoke a method of the corresponding courier or alternatively the courier could call an administrator's method when the reply becomes available.

The former alternative has the disadvantage that the administrator may get blocked if it invokes the courier too early. This may never occur with the latter approach. However, the administrator has to manage local queues for replies that are sent to it and that it cannot use immediately. Furthermore, each time a reply is returned, it should check whether all the replies needed so far for handling a client's request are available so that it may proceed with the client's request.

The use of proxy objects for carrying out requests and for storing replies is also needed in the case of one-way message passing for allowing requests to be paired with replies.

Although proxies are a general programming approach, it is cumbersome to program and use them explicitly. In fact unless the language supports classes with type parameters and a flexible manipulation of method names, a new proxy class would have to be defined for each different worker class in an administrator application.

Future variables in ABCL/1 [94], the *process* type in PAL [18] and CBox objects in ConcurrentSmalltalk [92] provide functionality which is somewhat similar to the courier proxies that were used by the administrator to call workers. These mechanisms could be used by the administrator to call workers without getting blocked and for collecting worker replies at a later time.

The advantage of these mechanisms over program-defined proxies is that they can be used for calling workers of any class. Future variables, however, are not first-class objects and so are not as flexible. For instance, a future variable cannot be sent in a message allowing a different object than the one that made the request to receive the reply.

A difficulty with built-in proxies is that the administrator may at some point in time have to get blocked and wait for a further client request or the reply to a previous worker request. Unless there exists a synchronization mechanism that allows the administrator to wait on either of these events, the administrator may get blocked to obtain a reply or request that is not available and will thus be unable to accept other requests or replies. This problem could be circumvented either by polling if a non-blocking request acceptance mechanism is supported or by additional, explicitly programmed proxies that would return the replies by invoking some administrator's operation especially provided for that purpose. This way a synchronization mechanism for selectively accepting requests would allow the administrator to be woken up either for receiving the results of a previous requests or for accepting new requests.

Still, the administrator's code may get quite involved. If there is no way to prevent being woken up by messages containing client requests or worker replies that cannot be used right away, local message queues will have to be managed by the administrator. So, it appears that built-in proxies combined with single-thread objects provide limited support for reply scheduling by the administrator since one should again rely on the use of explicitly programmed proxies.

Combining Request/Reply and One-Way Message Passing

It is also possible to relax the RPC style of communication without going all the way to supporting one-way message passing as the main communication primitive. This has the advantage that it is possible to present an RPC interface to clients and, at the same time, obtain more flexibility for processing requests by the administrator. This possibility is illustrated by ABCL/1 [94] which permits the pairing of an RPC interface at the client side with one-way asynchronous message passing at the administrator's side. Moreover, the reply message does not have to be sent by the administrator object. This provides even more flexibility in the way that the administrator may handle requests since the replies may be directly returned to the client by proxies. The following segment of code shows how this is accomplished.

The RPC call at the client side looks like:

```
result := [ administrator <== :someRequest arg1 ... argn] ...
```

A message is sent to the administrator to execute the request someRequest with arguments arg1,...,argn. The client is blocked until the reply to the request is returned and the result is stored in the client's local variable result.

At the administrator's side the client's request is accepted by matching the message pattern:

```
(=> :someRequest arg1 ... argn @ whereToReply
      .... actions executed in response to this request ... )
```

When the administrator accepts this request, the arguments are made available in the local variables arg1,...,argn and the *reply destination* of the request in the local variable

whereToReply. The reply destination may be used as the target of a "past type," i.e. asynchronous, message for returning the reply to the client. As a reply destination may also be passed around in messages, it is possible for another object to send the reply message to the client. This action would look like:

```
[ whereToReply <== result ]
```

where whereToReply is a local variable containing the reply destination obtained by the message acceptance statement shown above, and result is the result of the client's request.

Another interesting way of using the possibility to combine one-way message passing with RPC is for flexible reply scheduling by the administrator. In the previous section, on built-in proxies, we mentioned that a difficulty was that the administrator should be able to wait to accept both returned replies and further requests. A way to circumvent this problem was to use explicitly programmed proxies that would return results by invoking some operation provided by the administrator. In this way, replies were returned by requests so that a request acceptance mechanism was sufficient for allowing the administrator to wait for both requests and replies. A different approach is possible by pairing one-way messages to the RPC interface supported by workers. With this approach, the administrator may use a past type message, with itself as reply destination, for calling the workers which present an RPC interface. The replies from the workers can then be received by the administrator as any past-type message request. This allows the administrator to use the message acceptance mechanism for receiving both requests and replies.

This approach has, however, some of the drawbacks of one-way message passing: some extra work is needed in order to find out which reply message is related to what request and also that the administrator has to manage queues for replies that may not be used immediately.

2.4.2.2 Multi-Threaded Objects

Another way for allowing the administrator to service several concurrent requests is by supporting multiple concurrent or quasi-concurrent threads. A separate concurrent thread may now be used for handling each client request. However, depending on the mechanisms provided for thread creation and scheduling, it may still be necessary to resort to the solutions discussed previously in order to achieve a satisfactory level of concurrency in the processing of client requests.

We consider in turn quasi-concurrent and concurrent approaches and examine the support provided by the thread creation and scheduling mechanisms for programming administrators.

Quasi-Concurrent Approaches

A traditional example of "objects" with quasi-concurrent thread structure is provided by monitors [42] [21]. However, monitors present some well-known difficulties such as "nested monitor calls," and they unduly constrain parallelism [56] [77] [20] when used as

the main modular units of concurrent programs. These limitations are due to some extent to the quasi-concurrent structure of threads. However, an approach based on monitors would also constrain concurrency among different objects because of its limited support for reply scheduling. Assuming that the administrator is a monitor, then when calling a worker the monitor would remain blocked until the invoked operation would return. This situation, called *remote delay* [53], makes it impossible for the administrator to accept further client requests or to call a second worker.

Consequently, certain object-oriented languages have adopted more flexible variations. For example, Emerald [19] uses monitors as defined by Hoare [42]. However, not all operations of an object have to be declared as monitor procedures and also several independent monitors may be used in the implementation of an object. *Lock blocks* and *wait queues* in Trellis/Owl [68] also allow for more flexible implementation schemes than if objects were identified to monitors. With this approach, however, objects in these languages are not quasi-concurrent any more.

The restricted support for concurrency among objects by monitors is not due to the quasi-concurrent structure of objects, but rather to the limited flexibility for reply scheduling. This is illustrated by the second quasi-concurrent approach we examine which by providing a more flexible reply scheduling scheme does not restrict concurrency among objects.

Hybrid [71] is another language which adopts a quasi-concurrent thread structure for objects. However, in contrast to monitors, the *delegated call* mechanism provides a more flexible reply scheduling approach that does not restrain concurrency among objects. The administrator may use the delegated call mechanism to invoke workers. In such a case a new thread may be activated in the administrator for processing another client request in the meantime.

The delegated call mechanism is satisfactory for allowing the administrator to accept further client requests while a worker is executing a previous request, thus providing support for concurrency among several client requests. However, it is of no help for allowing several workers to execute in parallel for a single client request.

This may only be done by using proxies for invoking the workers or by a construct for specifying the creation of a new quasi-concurrent thread. Such a construct was proposed in the original design of Hybrid. The newly created quasi-concurrent threads would resume each other by using delegated calls. This construct was not included in the prototype because it substantially increased the complexity of the rules for message acceptance.

Concurrent Objects

With concurrent threads it is straightforward to process several client requests concurrently by creating a new thread for processing each client request. Provided that satisfactory mechanisms are supported for constraining the creation and activation of concurrent threads, this does not result in the mutual exclusion problems of languages with an orthog-

onal object model. The concurrent execution that may take place is explicitly specified by the programmer and the scope of the potential interference of the concurrent threads is restricted to the state of a single object.

Provided that there is some way to suspend the execution of a concurrent thread or avoid its creation, languages that support concurrent threads provide adequate support for request scheduling and for internal concurrency to the extent that several client requests may be processed concurrently.

A different issue that is not necessarily addressed by the support for concurrent threads is the possibility to use concurrency for processing a single request. Unless the creation of multiple threads can be initiated by the object, the support for reply scheduling of concurrent threads is not sufficient for processing a request in parallel.

For example, the language Sina [84] makes it possible to use several concurrent threads within an object for processing requests; there is no direct means, however, for one of these threads to create more threads for calling the worker objects in parallel. This is done indirectly by creating a courier proxy, as described previously. It is therefore not necessarily redundant to support both multiple threads and non-blocking communication primitives.

A satisfactory way for calling workers in parallel without using proxies or asynchronous message passing is to support a construct by which more threads may be created in the object. In this case a worker can be called by each of these threads in an RPC fashion. With quasi-concurrent threads, a call to a worker should trigger the execution of another thread. In SR the code segment of the administrator that is used for issuing requests to workers in parallel would look like this:

```
co    result1 := w1.doWork(...) -> loadManager.terminated(w1)
//    result2 := w2.doWork(...) -> loadManager.terminated(w2)
oc
globalResult := computResult(result1,result2);
...
```

2.4.3 Inheritance and Reuse of Synchronization Constraints

A large body of research has concentrated on the issues of making effective use of inheritance in COOPLs as well as on the related issue of reusing synchronization constraints.We will provide a brief overview of this work. Then we will turn our attention to the issues discussed in section 2.3.2 and illustrate the issues and how they are addressed by various language designs putting particular emphasis on some points that have not received the attention they deserved in related work. More extensive presentations and systematic comparisons of the proposals for supporting inheritance and the reuse of synchronization constraints may be found in [63] [60] and [16].

2.4.3.1 A Brief Overview of Related Research

Eiffel// [26][27] and Guide [34][48] were two of the earliest proposals that attempted to combine inheritance and synchronization constraints by removing the constraints from the bodies of methods.

These approaches presented some shortcomings with respect to the ability to extend the synchronization constraints to account for new methods introduced by subclasses. The problems were independently identified by Kafura and Lee [45] and Tomlinson and Singh [83], who in turn proposed their own approaches for overcoming them. A common aspect of these proposals is that constraints are specified by associating sets of methods to abstractions of the object state in which they can be executed. The main idea was that the set of methods would be extended in subclasses with the additional methods.

Matsuoka *et al.* [62], however, showed that there existed certain cases, called *inheritance anomalies*, where new state abstractions would have to be introduced in subclasses, consequently requiring extensive redefinition of inherited methods. Matsuoka later proposed his own approach, where he retained the idea of sets of acceptable methods, and provided a combination of guards and accept sets allowing the best technique to be used for the problem at hand.

The issue of extending and combining inherited constraints was also addressed in various other proposals, notably: Synchronizing Actions [69], Scheduling Predicates [59], Ceiffel [57], Frølund's framework [37], PO [29], SINA [16] and SPN [74]. It is important to note that Synchronizing Actions and SPN are two of the very few proposals to consider the issue of suspending method execution, which is important for reply scheduling.

The language DRAGOON [13] [14] supports the specification of generic synchronization constraints and provides a special inheritance mechanism separate from class inheritance of sequential aspects of classes for reusing these synchronization constraints.

Meseguer [67] has proposed a somewhat different approach for avoiding the problems related to the use of inheritance in COOPLs. He proposes to eliminate the synchronization code which causes inheritance anomalies. His language is based on a concurrent rewriting logic; the use of appropriate rewrite rules allows the specification of synchronization without introducing inheritance anomalies.

Synchronizers [38] is an approach for the specification of synchronization constraints that allows constraints to be associated to objects dynamically. An interesting point about this proposal is that constraints may depend on the state and computation history of several other objects.

2.4.3.2 Binding Mechanisms for Synchronization Constraints

The most direct way to associate synchronization constraints to methods is to specify them together as part of a class definition. Constraints defined in a class are inherited by the ordinary class inheritance mechanism. Such an approach is followed by most COOPLs, such as Guide, PO, PROCOL and ACT++, to name a few. This approach, however, has the shortcoming that it may be difficult to apply constraints to different classes. A first problem is with method names: if constraints refer to particular method names of the class in which they are defined, it will be difficult to apply them to classes where meth-

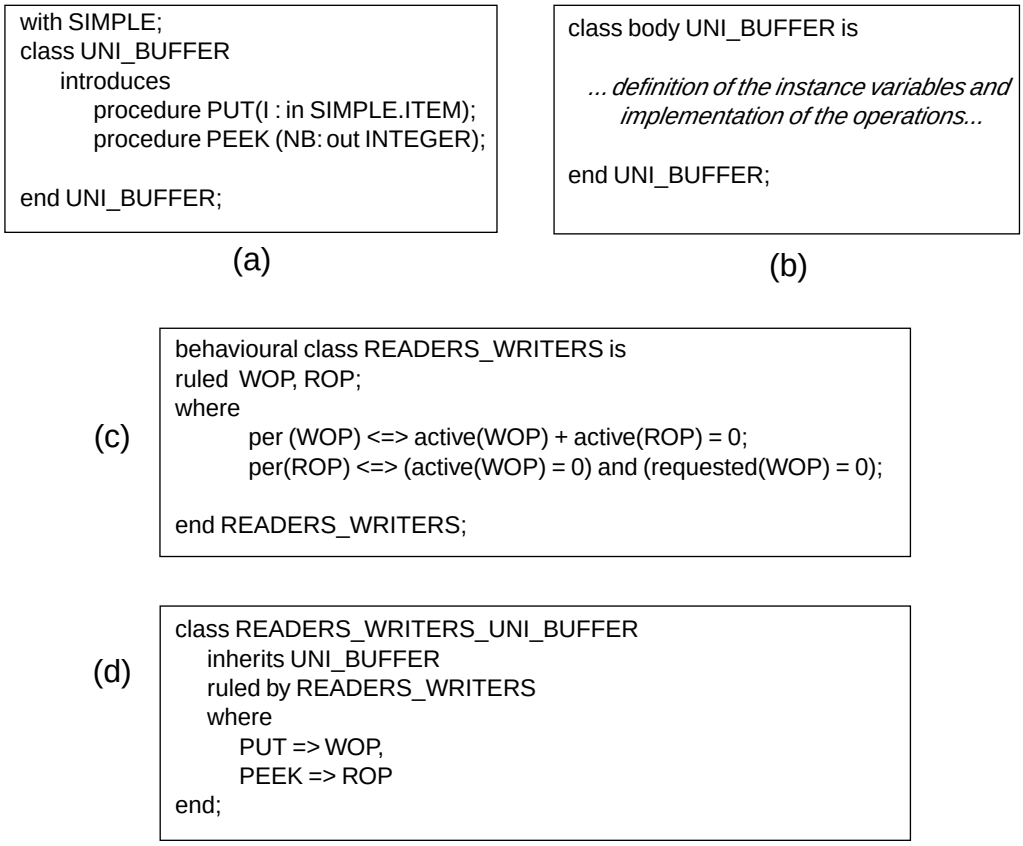

Figure 2.6 *Constraint definition in DRAGOON.*

ods have different names. Another problem comes from the use of class inheritance for reusing constraints. If one uses class inheritance to reuse the constraints, the methods defined in the class are also inherited. Below we examine some approaches that have been proposed for addressing these problems.

Genericity of Synchronization Constraints in DRAGOON

DRAGOON [13] [14] is an example of a language that supports the specification of generic synchronization constraints and of one that dissociates inheritance from the mechanism used for binding synchronization constraints to a class's methods. Generic constraints are defined as behavioural classes (b-classes). The constraints may be applied to a *sequential* class having no associated constraints, through the b-inheritance (behavioural) mechanism. This mechanism is independent from the inheritance mechanism (f-inheritance) used for sequential classes. Figure 2.6 shows an example of the use of the constraint definition and binding mechanism in DRAGOON. A class UNI_BUFFER is defined in (a) and (b) with methods PUT and PEEK used to insert a new element into the buffer and to examine the number of elements in the buffer. In (c) a generic constraint READERS_WRITERS

is defined for controlling execution of the methods of a class according to the readers, and writers, scheduling policy [81]. This synchronization constraint is bound to the class UNI_BUFFER in (d) where PUT is associated with the constraints for writers and PEEK with the ones for readers.

Using the Inheritance mechanism of Beta

A similar effect for specifying and binding constraints may be achieved by using the *inner* mechanism of Beta. In Beta a method in a subclass is associated with the superclass method it specializes. Instead of the subclass method explicitly invoking the superclass method through the use of *super* mechanism, as in Smalltalk, the superclass method is *always* invoked, and subclasses may only introduce additional behaviour at the point where the keyword inner occurs. In a sense, the execution of the superclass method is wrapped around the invoked subclass method. First are executed the actions in the superclass method that precede inner, then the subclass method is executed, then the actions of the superclass method that follow inner are executed.

This feature may be combined with low-level synchronization mechanisms, such as semaphores, to implement classes that encapsulate generic synchronization policies that can be applied to methods defined in subclasses in a way similar to how it is done in DRAGOON.

Assume there is a class ReaderWriterSched (not shown) with methods reader and writer that use semaphores to implement a reader/writer scheduling policy for the methods reader and writer. This synchronization may be applied to a class SynchedBuffer with operations empty, get, put as follows:

```
SynchedBuffer: @ |   ReaderWriterSched
(# .... instance variables....
        peek:       Reader(# ...implementation of peek... #)
        get:        Writer(# ...implementation of get... #)
        put:        Writer(# ....implementation of put..#)
#)
```

This allows the execution of peek to be constrained according the synchronization constraints of a reader, whereas get and put are synchronized according to the synchronization constraints that apply to writers. More on the use of inheritance in Beta to define generic synchronization policies can be found in [58].

Method Sets and Abstract Classes in ABCL/AP100

The *method set* feature provided in this language may be combined with abstract classes to define generic synchronization constraints that can be applied to several classes. Method sets are specified as part of class definitions, and are associated with synchronization constraints. Method sets can be inherited and modified in subclasses. Systematic use of

methods sets solves the problem of applying constraints to classes with different method names. The possibility of combining method sets with abstract classes (classes where not all methods are defined) can be used to provide facilities similar to those of DRAGOON. Abstract classes, making systematic use of method sets in synchronization constraints, can be used to represent generic constraints similar to DRAGOON's b-classes. However, in contrast to DRAGOON, programmers have to use the features provided by the language in a disciplined way. Another interesting feature of this language, discussed below, is that it is possible to combine synchronization constraints.

2.4.3.3 Polymorphism and Synchronization Constraints

Polymorphism of synchronization constraints is concerned with the potential applicability of constraints to different classes provided that the language supports an appropriate binding mechanism. There are two potential deficiencies with respect to this issue in approaches for specifying synchronization. The first is related to the use of instance variables in conditions constraining the activation of methods. The second concerns the use of constraints that specify mutual exclusion among methods in languages that support intra-object concurrency.

The first deficiency, also discussed by Bergmans [16], occurs in the proposals of Frølund [37] and Matsuoka [63], and in Guide and PROCOL, to cite a few examples. In these languages the conditions that are used in their constraints reference the object's instance variables. This makes it difficult to apply the constraints to classes implemented in a way that does not require these instance variables. Moreover, it makes it difficult to change the implementation of a class without having to consider the instance variables referenced in the constraints and, eventually, modifying the constraints as well. The problem may also be more severe than just modifying the constraints of a single class, as the constraints to be modified may be used by other subclasses as well. This could cause the re-examination and adjustment of the constraints of several subclasses of the class that was modified.

Two approaches have been be followed for alleviating this problem. First, instead of accessing directly the instance variables, conditions could be specified through a function that accesses the object state indirectly. If the implementation had to be modified, only these functions would need to be modified to account for the changes in the object state. This approach is followed for this precise reason by Sina in the way conditions are specified in wait filters [16] as well as in the specification of state predicates [74]. A second approach is to use condition variables to maintain an abstract state that is separate from the actual class implementation and is used purely for synchronization purposes. This approach is followed Synchronizing Actions, DRAGOON and PO.

The second potential deficiency occurs in languages with intra-object concurrency. In several languages with intra-object concurrency, such as Guide, DRAGOON and PO, synchronization constraints specify mutual exclusion properties among methods. The main reason for imposing mutual exclusion constraints on method executions is that method implementations access common instance variables. However, a different or modified implementation of a class may use a different set of instance variables and may have different

needs for mutual exclusion. Consequently, constraints that specify mutual exclusion properties among methods may find limited applicability to classes with a different implementation. Also, modifying the implementation of a class to which such constraints are attached, as discussed above for guards that reference instance variables, may cause the modification of the constraints attached to several classes. This problem, however, has not received any attention by other work in the area.

2.4.3.4 Extensibility and Compositionality

In languages such as DRAGOON, the issue of combining synchronization constraints is avoided by the way the language is designed; inheritance is not allowed among classes that are associated with synchronization constraints, r-classes, or the classes (b-classes) that are use to describe the constraints themselves. This approach has advantages and disadvantages. The separation of constraints from classes allows the use of inheritance between f-classes without having to be concerned how the associated constraints would have to be combined. The disadvantage is that there is no support for reusing constraints in the definition of new ones.

In other languages the issue of combining constraints is addressed either because class inheritance mechanism is tight up to the constraint binding mechanism or to allow constraints to be defined incrementally.

Frølund [37] proposed an approach for combining constraints of a class with those introduced in subclasses based on the view that constraints should become stricter in subclasses. The proposed approach for combining constraints supports this view by incrementally combining conditions that disable method execution. This way conditions may only become more strict in subclasses.

Matsuoka [63] provides a more elaborate way of combining constraints through modification of *method sets* and by the fact that method sets are recomputed in a subclass taking into account the full set of methods including the methods inherited from all superclasses. For instance, the method set all-except(LOCKED), where LOCKED is another method set defined elsewhere, denotes all the object's methods except the ones in LOCKED. This method set is recomputed in subclasses to account for additional methods defined in the subclass or inherited from other superclasses. Such features enable the definition of mixins that can be combined with the constraints of other classes to obtain the synchronization behaviour specified by the mixin. An example of such a mixin class is presented in [63].

A powerful way of composing synchronization constraints is also provided by wait filters in Sina. In order to get accepted, messages are matched against patterns of *wait filters*. Wait filters are associated with conditions, a form of guards, that must be true to let matching messages go through the filter. Filters can be stacked at the interface of an object and messages have to traverse all of them before being accepted by the object. Bergmans shows in [16] how this approach can be used for the locking mixin and for other constraint composition examples. The locking mixin discussed above can be realized by a class that provides a wait filter that matches all messages but unlock and is associated with a condition, Unlocked, that is true only when the object is unlocked. Lock and Unlock methods change the state of a lock object so as to render the Unlock condition false and true respec-

tively. A lock object can be used in the definition of another class in such a way that messages have to go through its filter first. In this way the synchronization constraint defined by lock can be reused in other classes.

PO [29] also supports the composition of constraints defined in superclasses of a class. In contrast to the proposals of Frølund and Matsuoka, where objects are single-threaded, PO constraints are designed for objects with internal concurrency. Constraints on the parallel execution of methods are partially ordered in a lattice with fully parallel execution of methods at the top and mutual exclusion among all methods at the bottom of the lattice. When incompatible constraints are inherited from different superclasses, they are compared according to this order and the more strict constraint is retained.

2.4.3.5 Combining Inheritance with Request/Reply Scheduling

In most work on the design of mechanisms for the specification and reuse of synchronization constraints, little attention has been paid to the eventuality that methods may have to be suspended halfway through their execution. However, as we discussed in section 2.4.2 this may be necessary to support reply scheduling. The possibility of suspending methods using mechanisms designed for the reuse of synchronization constraints is addressed in Synchronizing Actions [69] and in the design of the *state predicate* [74] mechanism.

Synchronizing Actions are based on multi-thread objects. The execution of a method may be suspended by calling, through *self*, another method with a pre-action such that the call is delayed. This approach may be used to support request and reply scheduling for the administrator as shown in figure 2.7. The administrator calls workers by creating proxy objects that do the actual call. After creating a proxy the administrator thread is suspended by calling the method suspend. The proxy calls the worker and when the call returns it calls the workerDone method to cause the administrator thread to be resumed. Figure 2.7 illustrates the implementation of the administrator concentrating on the synchronization aspects. Other languages that support internally concurrent objects and flexible specification of synchronization constraints, for instance Guide or Sina, could be used in a similar way. This approach, however, has some shortcomings. First, its complexity would make it difficult to use in practice. Second, it relies on the assumption that methods invoked through self are subject to the same constraints as invocations from other objects. This may not be appropriate when self is used in conjunction with inheritance to reuse algorithms defined in abstract superclasses.

The state predicate approach [74] provides a simpler and more direct way for suspending method execution based on a state predicate. The effect is similar to the one achieved by the approach discussed above. However, the resulting code is simpler as thread suspension and resumption is supported by the language and the complications deriving from the need to call the objects methods through self are avoided.

2.4.4 Summary

Below we present our observations with respect to reuse issues resulting from our exploration of language design approaches.

```
class Admin;
concurrency_control:
    boolean worker_finished := false,
            admin_idle := true;

method suspend()
    matching (true)
    pre { admin_idle := true }
    action{
        self!waitWorker ()
     }
    post { admin_idle := false}

method waitWorker()
    matching (worker_finished );
    pre { worker_finished := false;admin_idle := false
    }
    action { }
    post { };

method workerDone()
    matching (true)
    pre { worker_finished := true }
    action { }
    post { }

method request()
    matching ( admin_idle )
    pre { admin_idle := false}
    action {
        do some local processing...
        request := worker_proxy.doWork();
        self!waitWorker ();
        ...some more processing...
    }
    post { admin_idle := true };
```

Figure 2.7 *Request/reply scheduling with synchronization constraints.*

Object-Based Features

- *Homogeneous object models promote reuse:* Concurrent applications can safely reuse objects developed for sequential applications; efficiency need not be sacrificed.
- *Sequential objects with strict RPC are inadequate:* Request scheduling and internal concurrency can only be implemented by sacrificing the RPC interface; the solution is either to support concurrent threads or to relax the strict RPC protocol.
- *One-way message passing is expressive but undesirable*: Since higher-level request-reply protocols must be explicitly programmed, development and reuse of objects is potentially more error-prone.
- *Acceptance of concurrent requests is handled well either by concurrent threads or by explicit request/reply scheduling.*
- *Issuing concurrent requests is handled well by one-way message passing, by proxies or by internal concurrency:* The combination of both concurrent threads and non-blocking communication primitives may be appropriate for handling the separate issues of accepting and issuing concurrent requests.
- *Built-in proxies* used by *sequential objects* with non-blocking request issuing mechanisms provide adequate support for reply scheduling but are weak at combining reply and request scheduling.
- *Both concurrent objects and multi-object approaches are useful for internal concurrency:* These approaches for internal concurrency are both useful for different purposes. Concurrent threads make it easy to implement objects that may service several concurrent requests that do not modify the objects state. Multi-object approaches are

interesting when the implementation of a new object class, with internal concurrency, may be realized by using several concurrently executing instances of existing object classes.

Inheritance and Synchronization Constraints

- *Synchronization constraints should not be hardwired in methods:* If the synchronization code that schedules the execution of methods is hardwired in methods, it will be necessary to modify the method code in order to meet the constraints of other classes.
- *Multiple threads are needed to cope with reply scheduling:* To support reply scheduling it is important to be able to suspend the execution of a method. However, it seems difficult to do this if synchronization code is kept separate from methods to support inheritance.
- *Method suspension and resumption should be taken into account by synchronization constraints*: Taking into account the suspension of method execution by the mechanism that implements the synchronization constraints makes it simpler to program reply scheduling problems without compromising the reusability of methods.
- *Specification of mutual exclusion may lead to non-polymorphic constraints*: Mutual exclusion properties of methods are often related to the way methods access instance variables. Such constraints may thus not be applicable to classes with different instance variables or in which methods access instance variables in a different way. Including mutual exclusion specifications in constraints makes them less reusable.
- *It is advantageous to separate the reuse of constraints from inheritance*. It is easier to reuse synchronization constraints is they are specified generically and if their application to different classes is not accomplished through class inheritance.

2.5 Conclusion

Integrating concurrency and object-oriented programming is not as easy as it may seem at a first sight. There is no major difficulty in introducing both object-oriented and concurrency features in a single language. However, arbitrary combinations of concurrency and object-oriented features do not allow programmers draw the benefits of object-oriented programming for the development of concurrent systems. These difficulties have fostered substantial research in the past few years in the design of languages that gracefully integrate both kinds of features. However, the interference of the features occurs in several aspects of language design and the various proposals are not equally successful in all these aspects.

In this chapter we have discussed a number of issues that should be addressed in various aspects of language design, and we have formulated some criteria to use in evaluating design choices. We have used these criteria to evaluate various proposals, and we have illustrated the issues by examining specific languages. The languages discussed were chosen to illustrate particular points rather than to present a complete survey of all existing pro-

posals. It was not our intention to compare individual languages; other issues not discussed in this chapter would have to be considered in such an endeavour. Different considerations come in to play, for example, when designing a language for rapid prototyping or a language for programming embedded systems.

We have presented some guidelines for the design of languages that support the basic object-oriented features promoting reuse. Although these seem to be necessary conditions more is needed to achieve reuse at a larger scale. These are research issues which are discussed in other chapters. The further development and the use of techniques for reuse at a larger scale for developing concurrent systems may provide more criteria for evaluating language features and may result in more requirements on language design.

References

[1] Gul Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass., 1986.

[2] Gul Agha and C. J. Callsen, "ActorSpace: An Open Distributed Programming Paradigm," *Proceedings 4th ACM Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices*, vol. 28, no. 7, 1993, pp. 23–323

[3] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass., 1986.

[4] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans and Akinori Yonezawa, "Abstracting Object Interactions Using Composition Filters," *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming*, ed. R. Guerraoui, O. Nierstrasz, M. Riveill, *Lecture Notes in Computer Science*, vol. 791, Springer-Verlag, 1994, pp. 152–184

[5] Pierre America, "Inheritance and Subtyping in a Parallel Object-Oriented Language," *Proceedings ECOOP '87*, ed. J. Bézivin, J-M. Hullot, P. Cointe and H. Lieberman , *Lecture Notes in Computer Science*, vol. 276, Springer-Verlag, Paris, 1987, pp. 234–242.

[6] Pierre America, "POOL-T: A Parallel Object-Oriented Language," in *Object-Oriented Concurrent Programming*, ed. A. Yonezawa and M. Tokoro, MIT Press, Cambridge, Mass., 1987, pp. 199–220.

[7] Pierre America, "A Behavioural Approach to Subtyping in Object-Oriented Programming Languages," in *Proceedings of the Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Viareggio, Italy, Feb. 1989, pp. 141–156.

[8] Pierre America and Frank van der Linden, "A Parallel Object-Oriented Language with Inheritance and Subtyping," *Proceedings OOPSLA'90, ACM SIGPLAN Notices*, vol. 25, no. 10, ACM Press, Oct. 1990, pp. 161–168.

[9] American National Standards Institute, Inc., *The Programming Language Ada Reference Manual, Lecture Notes in Computer Science*, vol. 155, Springer-Verlag, 1983.

[10] S. Andler, "Predicate Path Expressions," in *Proceedings of 6th ACM POPL, ACM SIGPLAN Notices*, 1979.

[11] Gregory R. Andrews and Fred B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, vol. 15, no. 1, March 1983, pp. 3–43.

[12] Gregory R. Andrews, R.A. Olsson and M. Coffin, "An Overview of the SR Language and Implementation," *TOPLAS*, vol. 10, no. 1, Jan. 1988, pp. 51–86.

[13] Colin Atkinson, Stephen Goldsack, Andrea Di Maio and R. Bayan,"Object-Oriented Concurrency and Distribution in DRAGOON," *Journal of Object-Oriented Programming*, March/April 1991.

[14] Colin Atkinson, *Object-Oriented Reuse, Concurrency and Distribution*, Addison-Wesley/ACM Press, 1991.

[15] Henri E. Bal, J.G. Steiner and Andrew S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, vol. 21, no. 3, Sept. 1989, pp. 261–322.

[16] Lodewijk Bergmans, "Composing Concurrent Objects," Ph.D. Thesis, University of Twente, 1994.

[17] Ted Biggerstaff and C. Richter, "Reusability Framework, Assessment and Directions," *IEEE Software*, vol. 4, no. 2, March 1987, pp. 41–49.

[18] Anders Bjornerstedt and Stefan Britts, "AVANCE: An Object Management System," *Proceedings OOPSLA'88, ACM SIGPLAN Notices*, vol. 23, no. 11, San Diego, Nov. 1988, pp. 206–221.

[19] Andrew Black, Norman Hutchinson, Eric Jul and Henry Levy, "Object Structure in the Emerald System," *Proceedings OOPSLA'86, ACM SIGPLAN Notices*, vol. 21, no. 11, Nov. 1986, pp. 78–86.

[20] Toby Bloom, "Evaluating Synchronisation Mechanisms," in *Proceedings of the Seventh Symposium on Operating System Principles*, ACM-SIGOPS, Dec. 1979.

[21] Per Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, vol. SE-1, June 1975, pp. 199–207.

[22] Jean-Pierre Briot and Akinori Yonezawa, "Inheritance and Synchronisation in Concurrent OOP," *Proceedings ECOOP 87*, Paris, June 1987, BIGRE, no. 54, June 1987, pp. 35–43.

[23] Jean-Pierre Briot, "Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment," in *Proceedings ECOOP 89*, ed. S. Cook, British Computer Society Workshop Series, Cambridge University Press, 1989.

[24] Roy H. Campbell and A.Nico Habermann, "*The Specification of Process Synchronisation by Path Expressions*," *Lecture Notes in Computer Science*, vol. 16, Springer-Verlag, New York, 1974, pp. 89–102.

[25] Luca Cardelli and Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, Dec. 1985, pp. 471–523.

[26] Denis Caromel,"A General Model for Concurrent and Distributed Object-Oriented Programming," *Proceedings ACM SIGPLAN OOPSLA 88 workshop on Object-Based Concurrent Programming, ACM SIGPLAN Notices*, vol. 24, no. 4, April 1989, pp. 102–104.

[27] Denis Caromel, "Concurrency and Reusability: From Sequential to Parallel," *Journal of Object-Oriented Programming*, Sept./Oct. 1990.

[28] William Cook, "A Proposal for Making Eiffel Type-Safe," in *Proceedings ECOOP 89*, ed. S. Cook, British Computer Society Workshop Series, Cambridge University Press, 1989.

[29] Antonio Corradi and L. Leonardi, "Parallelism in Object-Oriented Programming Languages," *Proceedings of IEEE International Conference on Computer Languages*, March 1990, New Orleans, IEEE Computer Society Press, pp. 261–270.

[30] P. Courtois, F. Heymans and D. Parnas, "Concurrent Control with Readers and Writers," *Communications of the ACM*, vol. 14, no. 10, Oct. 1971, pp. 667–668.

[31] Brad J. Cox, *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.

[32] Stefano Crespi Reghizzi, G. Galli de Paratesi and S. Genolini, "Definition of Reusable Concurrent Software Components," *Lecture Notes in Computer Science*, vol. 512, Springer-Verlag, July 1991, *Proceedings of ECOOP 91*, Geneva, pp. 148–166.

[33] S. Danforth and Chris Tomlinson, "Type Theories and Object-Oriented Programming," *ACM Computing Surveys*, vol. 20, no. 1, March 1988, pp. 29–72.

[34] Dominique Decouchant, Sacha Krakowiak, M. Meysembourg, Michel Rivelli and X. Rousset de Pina, "A Synchronisation Mechanism for Typed Objects in a Distributed System," *Proceedings ACM SIGPLAN OOPSLA 88 workshop on Object-Based Concurrent Programming, ACM SIGPLAN Notices*, vol. 24, no. 4, April 1989, pp. 105–107.

[35] L. Peter Deutsch, "Reusability in the Smalltalk-80 Programming system," in *IEEE Tutorial on Software Reusability, 1987*.

[36] L. Peter Deutsch, "Design Reuse and Frameworks in the Smalltalk-80 system," in *Software Reusability*, ed. T. J. Biggerstaff and A. J. Perlis, vol. 2, ACM Press, 1989, pp. 57–71.

[37] Svend Frølund, ''Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages,'' *Proceedings ECOOP 92*, ed. O. Lehrmann Madsen, *Lecture Notes in Computer Science*, vol.615, Springer-Verlag, Utrecht, June/July 1992, pp. 185–196.

[38] Svend Frølund and Gul Agha, "A Language Framework for Multi-Object Coordination," *Proceedings ECOOP'93*, *Lecture Notes in Computer Science*, vol. 707, July 1993, pp. 346–360.

[39] Morven Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," *Software—Practice and Experience*, vol. 11, 1981, pp. 435–466.

[40] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[41] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, Feb. 1972, pp. 271–281.

[42] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, vol. 17, no. 10, Oct. 1974, pp. 549–557.

[43] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, Aug. 1978, pp. 666–677.

[44] Ralph E. Johnson and Brian Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, June/July 1988, pp. 22–35.

[45] Dennis G. Kafura and Kueng Hae Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages," in *Proceedings ECOOP'89*, ed. S. Cook, British Computer Society Workshop Series, Cambridge University Press, 1989.

[46] Alan H. Karp, "Programming for Parallelism," *IEEE Computer*, 1987, pp. 43–577

[47] Dimitri Konstantas, Oscar M. Nierstrasz and Michael Papathomas, "An Implementation of Hybrid," in *Active Object Environments*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, 1988, pp. 61–105.

[48] Sacha Krakowiak, M. Meysembourg, H. Nguyen Van, Michel Riveill, C. Roisin and X. Rousset de Pina, "Design and Implementation of an Object-Oriented Strongly Typed Language for Distributed Applications," *Journal of Object-Oriented Programming*, vol. 3, no. 3, Sept./Oct. 1990, pp. 11–22

[49] Chris Laffra, "PROCOL: A Concurrent Object Language with Protocols, Delegation, Persistence and Constraints," Ph.D. thesis, Erasmus University, Rotterdam, 1992.

[50] Butler W. Lampson and D.D. Redell, "Experience with Processes and Monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, 1980, pp. 105–117.

[51] Henry Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," *Proceedings OOPSLA '86*, *ACM SIGPLAN Notices*, vol. 21, no. 11, Nov. 1986, pp. 214–223.

[52] Barbara Liskov, Alan Snyder, Robert Atkinson and Craig Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, vol. 20, no. 8, Aug. 1977, pp. 564–576.

[53] Barbara Liskov and S. Zilles, "Programming with Abstract Data Types," *Proceedings of the ACM Symposium on Very High Level Languages*, *ACM SIGPLAN Notices*, vol. 9, no. 4, 1974, pp. 50–59.

[54] Barbara Liskov, Maurice Herlihy and L. Gilbert, "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing," in *Proceedings of the 13th ACM POPL*, St Petersburg, Fla., 1986.

[55] Barbara Liskov, "Distributed Programming in Argus," *Communications of the ACM*, vol. 31, no. 3, 1988, pp. 300–313.

[56] A. Lister, "The Problem of Nested Monitor Calls," *ACM Operating Systems Review*, July 1977, pp. 5–7.

[57] Peter Löhr, "Concurrency Annotations for Reusable Software," *Communications of the ACM*, vol. 36, no. 9, Sept. 1993, pp.81–89.

[58] Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard, *Object-Oriented Programming in the Beta Programming Language*, Addison-Wesley, Reading, Mass., 1993.

[59] Ciaran McHale, Bridget Walsh, Sean Baker and Alexis Donnelly,"Scheduling Predicates," *Proceedings of the ECOOP'91 workshop on Object-Based Concurrent Computing*, ed. M. Tokoro, O. Nierstrasz and P. Wegner, *Lecture Notes in Computer Science* , vol. 612, 1992, pp. 177–193.

[60] Ciaran McHale, Bridget Walsh, Sean Baker and Alexis Donnelly, "Evaluating Synchronisation Mechanisms: The Inheritance Matrix," Technical Report, TCD-CS-92-18, Department of Computer Science, Trinity College, Dublin 2, July 1992, (presented at the ECOOP'92 Workshop on Object-Based Concurrency and Reuse).

[61] Pattie Maes, "Concepts and Experiments in Computational Reflection," in *Proceedings OOPSLA'87*, *ACM SIGPLAN Notices*, vol. 22, no. 12, Dec. 1987.

[62] Satoshi Matsuoka, Ken Wakita and Akinori Yonezawa, "Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages," (Extended Abstract), *Proceedings of OOPSLA/ECOOP'90 workshop on Object-Based Concurrent Systems*, *ACM SIGPLAN Notices*, 1990.

[63] Satoshi Matsuoka, Kenjiro Taura and Akinori Yonezawa, "Highly Efficient and Encapsulated Re-use of Synchronisation Code in Concurrent Object-Oriented Languages," *Proceedings OOPSLA'93*, *ACM SIGPLAN Notices*, vol. 28, no. 10, Oct. 1993, pp. 109–129

[64] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, New York, 1988.

[65] Bertrand Meyer, "Reusability: The Case for Object-Oriented Design," IEEE Software, vol. 4, no. 2, March 1987, pp. 50–64.

[66] Bertrand Meyer, "Systematic Concurrent Object-Oriented Programming," *Communications of the ACM*, vol. 36, no. 9, Sept. 1993, pp. 56–80.

[67] José Meseguer, "Solving the Inheritance Anomaly in Object-Oriented Programming," in *Proceedings ECOOP'93*, *Lecture Notes in Computer Science*, vol. 707, ed. O.M. Nierstrasz, Springer-Verlag 1993.

[68] J. Eliot B. Moss and Walter H. Kohler, "Concurrency Features for the Trellis/Owl Language," *Proceedings of ECOOP '87*, BIGRE, no. 54, June 1987, pp. 223–232.

[69] Christian Neusius, "Synchronizing Actions," *Proceedings of ECOOP'91*, *Lecture Notes in Computer Science*, vol. 512, Springer-Verlag, July 1991, pp. 118–132.

[70] Oscar Nierstrasz, "Active Objects in Hybrid," *Proceedings OOPSLA'87*, *ACM SIGPLAN Notices*, vol. 22, no. 12, Dec. 1987, pp. 243–253.

[71] Oscar Nierstrasz, "A Tour of Hybrid — A Language for Programming with Active Objects," *Advances in Object-Oriented Software Engineering*, ed. D. Mandrioli and B. Meyer, Prentice Hall, 1992, pp. 167–182.

[72] Michael Papathomas and Dimitri Konstantas, "Integrating Concurrency and Object-Oriented Programming: An Evaluation of Hybrid," in *Object Management*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, 1990, pp. 229–244.

[73] Michael Papathomas, "Concurrency Issues in Object-Oriented Languages," in *Object Oriented Development*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, 1989, pp. 207–245.

[74] Michael Papathomas, "State Predicate Notifiers: A Concurrent Object Model," Lancaster University Report, April 1994.

[75] David L. Parnas, "A Technique for Software Module Specification with Examples," *Communications of the ACM*, vol. 15, no. 5, May 1972, pp. 330–336.

[76] David L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, Dec. 1972, pp. 1053–1058.

[77] David L. Parnas, "The Non-Problem of Nested Monitor Calls," *ACM Operating Systems Review*, vol. 12, no. 1, 1978, pp. 12–14.

[78] Geoffrey A. Pascoe, "Encapsulators: A New Software Paradigm in Smalltalk 80," in *Proceedings of OOPSLA '86, ACM SIGPLAN Notices*, Sept. 1986.

[79] Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," ACM SIGPLAN Notices, vol. 21, no. 11, Nov. 1986, pp. 38–45.

[80] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[81] T. J. Teorey and T. B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM*, vol. 15, no. 3, March 1972, pp. 177–184.

[82] Tom Thompson, "System 7.5: A Step Toward the Future," *Byte*, August 1994.

[83] Chris Tomlinson and Vineet Singh, "Inheritance and Synchronisation with Enabled Sets," *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, vol. 24, no. 10, Oct. 1989, pp. 103–112.

[84] Anand Tripathi and Mehmet Aksit, "Communication, Scheduling, and Resource Management in Sina," *Journal of Object-Oriented Programming*, Nov./Dec. 1988, pp. 24–36.

[85] Jan Van Den Bos and Chris Laffra, "PROCOL: A Parallel Object Language with Protocols," *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, vol. 24, no. 10, Oct. 1989, pp. 95–102.

[86] Takuo Watanabe and Akinori Yonezawa, "Reflection in an Object Oriented Concurrent Language," ACM SIGPLAN Notices, vol. 23, no. 11, 1988, pp. 306–315.

[87] Peter Wegner, "Dimensions of Object-Based Language Design," in *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, Orlando, Florida, Dec. 1987, pp. 168–182.

[88] Peter Wegner and Stanley B. Zdonik, "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like," in *Proceedings ECOOP'88*, *Lecture Notes in Computer Science*, vol. 322, Springer-Verlag, 1988, pp. 55–77.

[89] Peter Wegner, "Concepts and Paradigms of Object-Oriented Programming," *ACM OOPS Messenger*, vol. 1, no. 1, August 1990.

[90] William E. Weihl, "Linguistic Support for Atomic Data Types," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 2, 1990.

[91] Rebecca J. Wirfs-Brock and Ralph E. Johnson, "Surveying Current Research in Object-Oriented Design," *Communications of the ACM*, vol. 33, no. 9, Sept. 1990, pp. 104–123.

[92] Yasuhiko Yokote and Mario Tokoro, "Concurrent Programming in ConcurrentSmalltalk," in *Object-Oriented Concurrent Programming*, ed. M. Tokoro, *MIT Press*, Cambridge, Mass., 1987, pp. 129–158.

[93] Yasuhiko Yokote and Mario Tokoro, "Experience and Evolution of ConcurrentSmalltalk," in *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, Orlando, Florida, Dec. 1987, pp. 168–182.

[94] Akinori Yonezawa, Etsuya Shibayama, T. Takada and Yasuaki Honda, "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1," in *Object-Oriented Concurrent Programming*, ed. M. Tokoro, *MIT Press*, Cambridge, Mass., 1987, pp. 55–89.