# Supporting Software Reuse in Concurrent Object-Oriented Languages: Exploring the Language Design Space[1]

Michael Papathomas
Oscar Nierstrasz[2]

## Abstract

The design of programming languages that cleanly integrate concurrency constructs and object-oriented features that promote software reuse is an open problem. We describe a design space that characterizes approaches to object-oriented concurrency in terms of a number of language design choices concerning the relationship between objects and concurrency. We identify requirements for software reuse and, with the help of an example that illustrates several of these requirements, explore the design space in order to evaluate which design choices interfere with reuse and which appear to adequately support it. We conclude by highlighting open research issues which, we believe, are essential for achieving effective reuse of concurrent software.

## 1. Introduction

The potential for interference between concurrency constructs and object-oriented mechanisms supporting reuse has been noted recently by a number of independent researchers [8], [17], [29], [32]. Whereas one might (naively) expect issues of concurrency to be orthogonal to those of encapsulation and reuse, it turns out that objects whose implementations make use of concurrency constructs are less likely to be extendible or reusable within new contexts without considerable overhead [28].

Our experience [29] designing and developing a COOPL, called Hybrid [24], has shown that some of the key questions concerning reuse are as follows:

- How easy is it to reuse objects developed in other applications? Is it necessary to understand the implementation of an object to correctly reuse it in a new context?
- To what extent is it possible to modify an object's implementation without affecting the rest of the application? Which implementation choices may be made freely without concern for clients' behaviour?
- Under what circumstances will an application's interaction structure be generic enough that it can be reused to produce new applications by "plugging in" independently developed objects and subsystems?
- Are the primitives for concurrency, communication and synchronization compatible with object-oriented reuse mechanisms such as class inheritance, late binding and pa-

1. In *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1991, pp. 189-204.

2. Authors address: Centre Universitaire d'Informatique, 12 rue du Lac, CH-1207 Geneva, Switzerland. E-mail: {michael, oscar}@cui.unige.ch. Tel: +41 (22) 787.65.80. Fax: +41 (22) 735.39.05.

rameterization? Is incremental modification of object classes difficult or impossible when concurrency is at stake?

Although a number of specific instances of interference between concurrency constructs and object-oriented features supporting reuse have been identified and remedies have been proposed, there exists as yet no general framework for evaluating language design with respect to support for reusability. As a step towards such a framework, in §2 we propose a design space that expresses language design choices in terms of the relationship between objects and concurrency. In §3 we identify a set of informal requirements for supporting reuse and we introduce a generic "administrator" example that illustrates the reuse issues. Language design choices are evaluated in §4 with respect to these requirements. We conclude with some remarks on open problems concerning software composition in a concurrent setting.

## 2. A Design Space for Concurrent OOPLs

We seek to evaluate language design choices with respect to the degree to which software reuse is supported. In particular, we wish to understand how choices of concurrency constructs 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?
- ***Internal Concurrency:*** can objects manage multiple internal threads?
- ***Client-Server Interaction:*** how much freedom and control do objects have in sending and receiving requests and replies?

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.

### 2.1 Concurrent Object Models

We shall first consider whether or not objects are provided with a default means of protecting internal consistency in the presence of concurrent requests. There are three main approaches:

- ***The Orthogonal Approach:*** Concurrent execution is independent of objects. Synchronization constructs such as semaphores in Smalltalk-80 [12], "lock blocks" as in Trellis/Owl [23]or monitors as in Emerald [6] 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 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 what method invocation it is willing to accept next. In POOL-T[2] this is specified by executing an explicit accept statement. In Rosette [32] an *enabled set* is used for specifying which set of messages the object is willing to accept next.

- ***The Heterogenous Approach:*** Both active and passive objects are provided. Passive objects do not synchronize concurrent requests. Examples of such languages are Eiffel // [10][22] and the language ACT++[17]. Both languages ensure that passive objects cannot be invoked concurrently by requiring that they be used only locally within single-threaded active objects.

Although most COOPLs fall clearly within one of these three categories, there are a number of limit cases. For example, Argus appears to support a heterogeneous model since it provides both *guardians* (active objects) and *clusters* (passive objects), but the synchronization of multiple threads within guardians corresponds to the orthogonal model.

### 2.2 Internal Concurrency

Wegner [36] 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 [37] and POOL-T and Ada tasks [1] are examples of sequential objects.

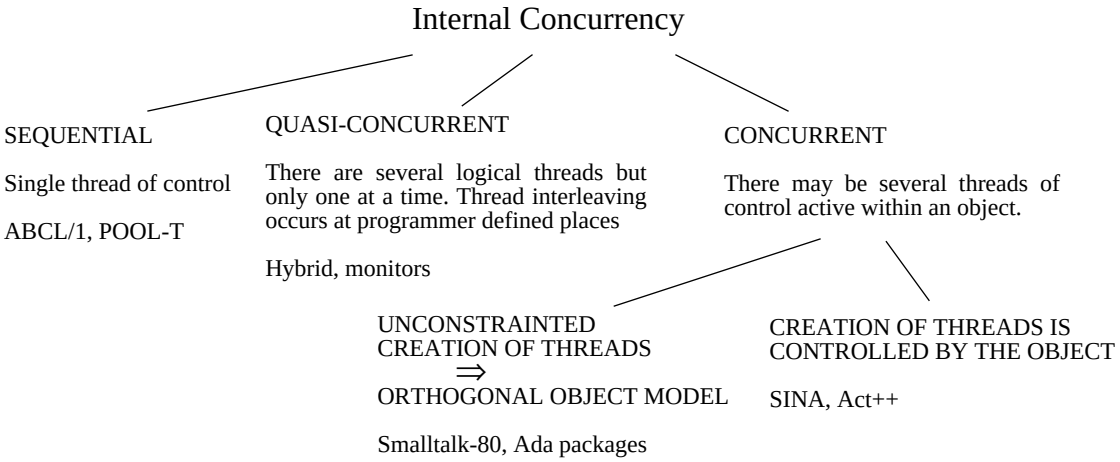
- ***Quasi-Concurrent Objects*** have multiple threads but only one thread may be active at a time. Control must be explicitly released to allow interleaving of threads. Hybrid domains[24] and monitors [15] are examples of such objects.

- ***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 in the sense that a new local thread is effectively created whenever a method is activated in response to a message.

We may further distinguish between concurrent objects that are capable of controlling the activation of multiple concurrent threads and those that cannot. For example, in the language SINA [33] a new concurrent thread may be created for the execution of method belonging to a select subset of the object's methods only if the currently active thread executes the *detach* primitive. The *next* and *become* primitives in Rosette and ACT++ can be used to achieve a similar effect, 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 [19], an object is associated with a set of activation conditions that specify which methods may be executed in parallel by internally concurrent threads.

### 2.3 Client-Server Interaction

In order to compare and contrast approaches to communication and synchronization, we view all concurrent object-oriented systems as collections of message-passing objects. By viewing the principal communication mechanism between objects as message passing we may focus on the interaction between objects playing client or server roles rather than on low-level issues concerning the interleaving and scheduling of threads of control within an object system. This will allow us to consider more easily issues of reuse from the perspective of clients and servers.




Initially, we may distinguish between languages providing one-way message passing primitives and those providing higher-level primitives that enforce balanced sending of request and reply messages.

- ***One-way Message Passing:*** the handling of *request* and *reply* messages must be explicitly programmed. Objects are not obliged to obey a request/reply protocol. Message passing may be synchronous or asynchronous.
- ***Request/Reply:*** communication primitives guarantee that requests will be eventually matched by replies. These primitives vary in the flexibility in sending and receiving messages they offer to clients and servers. Some approaches make use of "proxy" objects to disassociate the sending or receiving of messages from the current thread of control. Examples are *future variables*[37] and *CBoxes*[38].

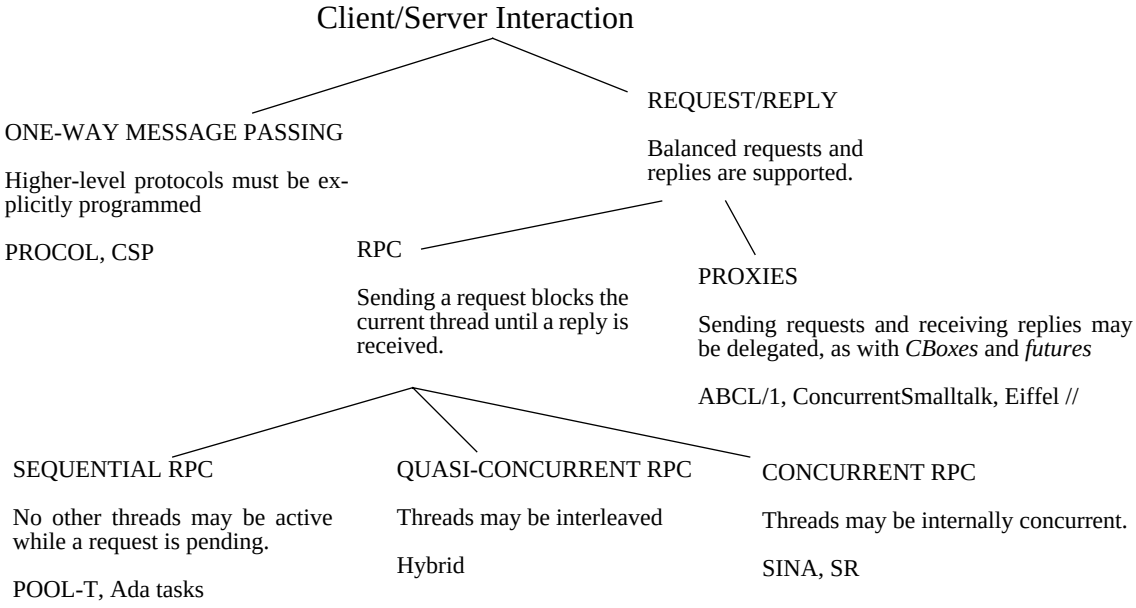
The degree of control that can be exercised by objects in the client and server roles allows us to further refine this initial classification. We specifically consider *reply scheduling*, which concerns the degree of flexibility the client has in accepting a reply, and *request scheduling*, which concerns the control the server can exercise in accepting a request.

### 2.3.1 The Client's View

From the client's point of view, there are three important issues concerning reply scheduling:

- ***Interleaving activities:*** Can the current thread continue after issuing the request? Alternatively, can another thread be active while the calling thread waits?
- ***Reply address:*** How and where is the reply to be sent? Flexible control over the reply destination can reduce the amount of message passing required.
- ***Getting the reply:*** What mechanisms are supported for matching replies to requests? How does the client synchronize itself with the computation and delivery of the reply?

As above, we distinguish between one-way message passing communication primitives and primitives supporting a request/reply protocol. Further flexibility in handling multiple requests and replies is obtained either by introducing *proxies* (i.e., external concurrency) or by introducing internal concurrency (or by a combination of these two):




### *One-way Message Passing*

Whether communication is synchronous, as in CSP [16] or PROCOL [34], 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 §4, 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.

Sequential (single-thread) RPC lacks flexibility in that clients are not able to submit multiple requests in parallel, since the client must wait for each reply in turn, and in that the server must reply to the client. (See also [20]). Although it is trivial to obtain a reply, it is not possible to interleave activities or to specify reply addresses.

Quasi-concurrent and concurrent clients obeying an RPC protocol are able to interleave concurrent activities either by transferring control to another request, as is possible with the *delegated call* mechanism of Hybrid [24], or by means of an explicit construct for initiating multiple concurrent threads as in SR [5].

***Proxies***

An alternative means to providing the client with more control 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* [37].) The actual client is therefore free to switch its attention to another activity in the mean time. The proxy itself may also perform additional computation or call multiple servers. Clearly, proxies work by manipulating the reply address to a request.

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 [10][37][38], 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 §4.

### 2.3.2 The Server's View

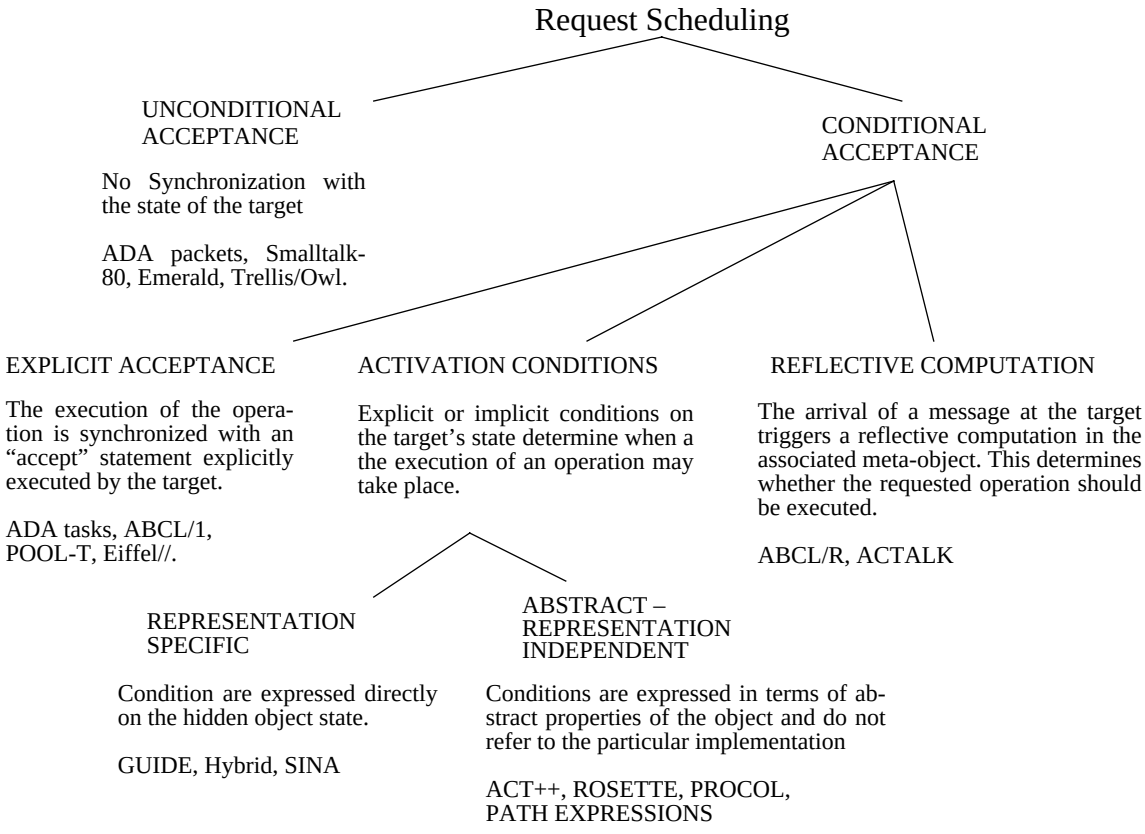
From the server's point of view the main concern is whether requests can be conditionally accepted[1]. When a request is sent the server may be busy servicing a previous request, waiting itself for another request to be fulfilled, or idle, but in a state that requires certain requests to be delayed. We distinguish initially between conditional and unconditional acceptance of requests. Conditional acceptance can be further discriminated according to whether requests are scheduled by an explicit acceptance, by activation conditions or by means of reflective computation:

*Unconditional acceptance* of requests implies an orthogonal object model and is characterized by the absence of synchronization with respect to the server's state.

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.

With *activation conditions*, requests are accepted on the basis of predicate over the message contents and the object's state. 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 on 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 references the object's instance variables, whereas in ACT++ the condition for accepting a message is that the object be at an appropriate *abstract state* which abstracts from

1. 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 [11], but this can also be seen as concern of internal concurrency where follow-up activity is viewed as belonging to a new thread.




the state of a particular implementation. Another approach is to specify the legal sequences of message acceptance by means of a grammar, as in path expressions and PROCOL [34].

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 [35] and it is also illustrated in Actalk [9] 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. Other languages also adopt a reflective model, however the synchronization is not expressed procedurally in the program of a meta-object.

## 3. Language Design Requirements for Reusability

The following requirements are motivated by the principle that reusable object classes should make minimal assumptions about the behaviour of applications that will use them.

1. ***Mutual Exclusion:*** 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

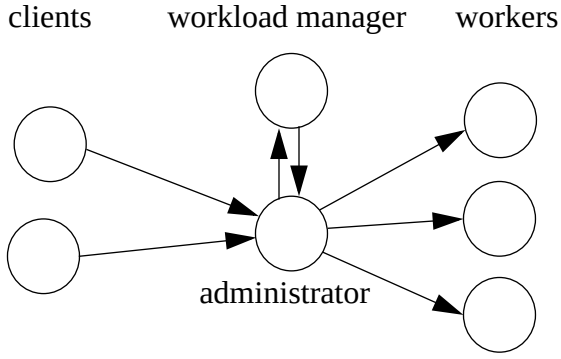
the client are not acceptable from the point of view of reusability since the client then cannot be written in a generic fashion.

3. ***Internal Concurrency:*** The concurrency constructs should allow for the implementation of objects that service several requests in parallel or are internally parallel for increased execution speed. This should be supported in a way that does not affect the clients so that sequential implementations of objects may be replaced by parallel ones. By the same token it should also be easy to coordinate the execution of already existing objects.

4. ***Reply Scheduling Transparency:*** A client should not be forced to wait until the serving object replies. In the mean time it may itself accept further requests or call other objects in parallel. It is also useful to specify that replies are to be sent to a proxy. (This is related to the problem of coping with *remote delays* [20], though we additionally consider that the client may initiate multiple requests in parallel.) Request 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.

5. ***Compositionality and Incremental Modification:*** Existing object classes should be reusable within new contexts without modification. Additionally, mechanisms for incremental modification of classes such as inheritance must be designed with special consideration given to concurrency to allow existing code to cooperate gracefully with modifications and extension [17], [32].

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 [11]. 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.




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 following aspects of the example are specifically related to reuse:

- *Mutual Exclusion:* (i) workload manager reuse – the workload manager must be protected from concurrent requests by the administrator (and its proxies); (ii) worker reuse – workers should similarly be protected as arbitrary objects may be used as workers;
- *Request Scheduling Transparency:* (iii) client reuse – the administrator must be able to interleave (or delay) multiple client requests, but the client should not be required to take special action 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 without special action by workers;
- *Compositionality:* (vi) administrator reuse – the administrator should be programmed in such a way that it can be reused by substituting only the part responsible for decomposing requests and composing replies (parameterization, inheritance, or other techniques may be appropriate).

There are other aspects of language design, apart from the concurrency features, that affect the creation and use of reusable administrator applications, such as the support for generic or parametrized classes and the support for dynamic binding. In our discussion however we concentrate on issues that are more specific to the concurrency features of languages and ignore those issues which would also arise in sequential languages.

## 4. Exploring the Language Design Space

In order to explore and evaluate the COOPL design choices we have selected, we shall consider in turn (i) object models, (ii) client/server interaction and internal concurrency, (iii) administrator reusability. Throughout, we shall refer to the reusability requirements of §3 and we will make use of the administrator example to illustrate specific points.

### 4.1 Concurrent 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 heterogenous 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.

## 4.2 Client/Server Interaction

As reusability is clearly enhanced when objects obey a standard protocol, we shall suppose that objects generally conform to a request/reply interface. If we provide an object model with communication primitives supporting only sequential RPC, we quickly discover that this is not enough to satisfy the requirements of the administrator. 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 (the request handler may make use of further "courier" proxies to call the workers). In this way the administrator is immediately free to accept new requests after returning the name of the request handler.

We must either relax the sequentiality constraint or the strict RPC protocol. The possibilities are: (i) one-way message passing, (ii) explicit request/reply scheduling primitives (with or without proxies), and (iii) internal concurrency. Let us consider each of these from the administrator's viewpoint.

### 4.2.1 Administrator Request Scheduling

#### *One-Way Message Passing*

An extreme solution in the direction of relaxing RPC is to support one-way synchronous or asynchronous message passing. In this case the administrator is free to accept messages and reply to them in the order it likes. One-way message passing has however some disadvantages.

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.

#### *Explicit Request/Reply Scheduling*

It is also possible to relax the RPC style of communication without going all the way to support 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 [37] which permits the pairing of 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. 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" 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 to the client's request.

***Internal Concurrency***

Another way for allowing the administrator to process several concurrent requests is to support multiple concurrent or quasi-concurrent threads. In Hybrid, for example, the administrator may issue requests to workers by *delegated calls*, thus temporarily suspending the calling thread and freeing the administrator to accept new requests from clients. In such a case a thread is used for handling each request. RPC-like communication can be used with clients since new threads can be created for processing requests even if the reply to other requests has not yet been returned.

### 4.2.2 Administrator Reply Scheduling

There are three main ways for the administrator to invoke the workers in parallel: (i) one-way message passing, (ii) proxies, and (iii) internal concurrency.

***One-Way Message Passing***

A difficulty with using one-way messages is getting the replies from workers. As there are several workers that are invoked in parallel and potentially concurrent invocations of single worker it is difficult for the administrator to tell which reply is associated with which request.

A solution to this problem is, for each request, to create a proxy which carries out the request. The proxy may then send a message to the administrator containing the worker's reply plus some extra information used for identifying the request. This solution has the advantage that the administrator will be triggered by the reply when it is available.

Another solution is for the administrator at some later time to call this object to obtain the result. In this case there can be a problem if the administrator blocks to obtain a result that is not

yet ready and thus ignores new client requests. This problem may be solved by introducing a non-blocking primitive for accepting requests, but reduces into polling the arrival of requests and replies. Polling could be avoided by a guarded command constructs where the guards could express both the acceptance of requests and arrival of replies.

***Proxies***

The administrator can call multiple workers in parallel by creating a number of "courier" proxies responsible for calling the workers and collecting the replies. The couriers may then be passed to another object responsible for composing the final result.

Future variables [37] and CBox [38] mechanisms provide functionality which is somewhat similar to courier objects. Future variables, however, are not first class objects and so are not as flexible since they cannot be sent in messages to other objects.

***Internal Concurrency***

In this case a construct is provided for the creation of concurrent or quasi-concurrent threads.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. Such constructs are provided in the original design of Hybrid [24] and in SR [5]. 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);
        ...
```

With such an approach it is not necessary to artificially decompose the administrator into multiple objects and proxies to obtain parallelism.

It should be noted that supporting multiple threads in this way is a different issue than using threads for servicing multiple client requests. For instance with the language SINA [33] it is 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 object, as described above. It is therefore not necessarily redundant to support both multiple threads and non-blocking communication primitives.

### 4.3 Administrator Reusability

We have concentrated thus far on reuse of objects without modification. For this it is clear that objects should support standard request/reply interfaces or other standard protocols. Problems of interference between concurrency and inheritance have been previously pointed by other researchers [17], [32], so we will only summarize and indicate some current trends.

Interference between existing code to be reused (e.g., superclasses) and incremental modifications (e.g., subclass extensions) is due to (i) the difficulty of the additional code to synchronize with the existing code and (ii) the difficulty of the existing synchronization code to be open to modifications yet to be defined. Kafura and Lee therefore propose an approach to synchroni-

zation based on explicit *abstract states* [17], though this approach does not adequately support request scheduling since activation conditions may not depend on message contents.

The main point, however, is that we lack good abstractions for incremental modifications to object classes. The message-passing interface should not be viewed in the same way as the inheritance interface seen by subclasses [31]. Current work in this area attempts to "unbundle" the mechanisms for software composition either in terms of software templates or "habitats" [30] or by decomposing inheritance into more primitive mechanisms [7], [14].

### 4.4 Summary

We can make the following initial observations concerning our exploration of reuse issues:

- *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 can only be implemented by sacrificing the RPC interface; the solution is to either permit internal concurrency 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 well-handled either by internal concurrency or by explicit request/reply scheduling.*
- *Issuing concurrent requests is well-handled by one-way message passing, by proxies or by internal concurrency:* the combination of both internal concurrency and non-blocking communication primitives may be appropriate for handling the separate issues of accepting and issuing concurrent requests.
- *Parallelism is better supported by internal concurrency than by distribution:* increasing parallelism by decomposing an object into concurrent subobjects and proxies introduces ad hoc dependencies between the parts; explicit mechanisms for managing concurrent threads seem to encourage better the design of reusable objects.

## 5. Conclusion and Future Directions

We have proposed and presented a design space for concurrent OOPLs that can be used to compare and evaluate various language design choices with respect to their support for software reusability. The design space distinguishes between (i) *object models* that provide varying degrees of concurrency control to objects, (ii) the degree of *internal concurrency* available to objects, and (iii) the flexibility and control objects can exercise during *client/server interactions.*

We have also identified a set of basic reusability requirements for concurrent objects, namely (i) a default level of *mutual exclusion* enhances server reusability, (ii) *request scheduling transparency* enhances client reusability by hiding server synchronization, (iii) *internal concurrency* should be transparent to both clients and servers, (iv) *reply scheduling transparency* en-

hances server reusability by hiding client synchronization, and (v) special consideration must be given to inheritance and other mechanisms for *composition* and *incremental modification* of reusable classes to avoid interference with concurrency constructs.

With the help of the example of an "administrator" framework that illustrates most of these reuse issues, we have explored the design space and drawn some initial conclusions. In particular, a homogeneous object model supporting an explicit request/reply protocol with internal request/reply scheduling together with constructs for managing multiple concurrent threads appears to encourage reuse better than other alternatives. Our observations are summarized in §4.4.

Our classification is far from exhaustive. In particular, we have emphasized imperative approaches to COOPL design, whereas declarative approaches based on logics appear to offer certain advantages with respect to abstraction of interface and implicit vs. explicit concurrency [3], [18], [21]. We have also been rather abstract in the presentation of our example. We feel that requirements for reusability should ultimately be based on practical experience in the development of reusable frameworks for concurrent applications. A standard set of test cases illustrating *both* expressiveness *and* reuse requirements is needed.

The most difficult open problem with respect to reuse of concurrent software is concerned with composition and incremental modification. At the level of language *mechanisms* a promising approach is to *unbundle* mechanisms like inheritance [7], [14] and think explicitly in terms of composition of software "patterns" [25], [26]. At the abstraction level, however, we lack good formalisms for reasoning about the abstract behaviour of concurrent objects. A step in this direction is to develop a notion of "plug compatibility" for the composition of objects based on the interactions between servers and clients [27]. The specification of the behavioural "contracts" [13] between collections of cooperating objects is a promising approach.

Finally, the earnest evaluation of language design choices depends on the ability to formalize the semantics of various language constructs and the ability to rapidly prototype and test the alternatives. We have developed a platform for the executable specification of COOPLs [25], [26] and we are proceeding with the study and evaluation of language designs.

## References

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

[2] P. America, "POOL-T: A Parallel Object-Oriented Language," in *Object-Oriented Concurrent Programming*, ed. A. Yonezawa, M. Tokoro, pp. 199-220, The MIT Press, Cambridge, Massachusetts, 1987.

[3] J-M. Andreoli and R. Pareschi, "LO and Behold! Concurrent Structured Processes," ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP '90, vol. 25, no. 10, pp. 44-56, Oct 1990.

[4] G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," ACM Computing Surveys, vol. 15, no. 1, pp. 3-43, March 1983.

[5] G.R. Andrews, R.A. Olsson and M. Coffin, "An Overview of the SR Language and Implementation," TOPLAS, vol. 10, no. 1, pp. 51-86, ACM, January 1988.

[6] A. Black, N. Hutchinson, E. Jul and H. Levy, "Object Structure in the Emerald System," ACM SIGPLAN Notices, Proceedings OOPSLA '86, vol. 21, no. 11, pp. 78-86, Nov 1986.

[7] G. Bracha and Wm. Cook, "Mixin-based Inheritance," ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP '90, vol. 25, no. 10, pp. 303-311, Oct 1990.

[8] J-P. Briot and A. Yonezawa, “Inheritance and Synchronization in Concurrent OOP,” Proceedings of the European Conference on Object-oriented Programming, pp. 35-43, Paris, France, June 15-17, 1987.

[9] J.P. Briot, “Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment,” in *Proceedings of the Third European Conference on Object-Oriented programming Languages*, ed. S. Cook, British Computer Society Workshop Series, Cambridge University Press, 1989.

[10] D. Caromel, “Concurrency and Reusability: From Sequential to Parallel,” JOOP, Sept./Oct. 1990.

[11] W.M. Gentleman, “Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept,” Software-Practice and Experience, vol. 11, pp. 435-466, 1981.

[12] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[13] R. Helm, I.M. Holland and D. Gangopadhyay, “Contracts: Specifying Behavioural Compositions in Object-Oriented Systems,” ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP ’90, vol. 25, no. 10, pp. 169-180, Oct 1990.

[14] A.V. Hense, “Denotational Semantics of an Object Oriented Programming Language with Explicit Wrappers,” Technical report A11/90, FB 14, Universität des Saarlandes, Nov. 5, 1990, (submitted for publication).

[15] C.A.R. Hoare, “Monitors : An Operating System Structuring Concept,” CACM, vol. 17, no. 10, pp. 549-557, ACM, October 1974.

[16] C.A.R. Hoare, “Communicating Sequential Processes,” CACM, vol. 21, no. 8, pp. 666-677, Aug 1978.

[17] D. G. Kafura and K. H. Lee, “Inheritance in Actor Based Concurrent Object-Oriented Languages,” in *Proceedings of the Third European Conference on Object-Oriented programming Languages*, ed. S. Cook, British Computer Society Workshop Series, Cambridge University Press, 1989.

[18] K.M. Kahn and V.A. Saraswat, “Actors as a Special Case of Concurrent Constraint Programming,” ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP ’90, vol. 25, no. 10, pp. 57-65, Oct 1990.

[19] S. Krakowiak et al.,”Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications,” JOOP, September/October 1990, pp. 11-22.

[20] B. Liskov, M. Herlihy and L. Gilbert, “Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing,” in *Proceedings of the 13th ACM symposium on Principles of Programming Languages*, St. Petersburg, Florida, 1986.

[21] J. Meseguer, “A Logical Theory of Concurrent Objects,” ACM SIGPLAN Notices, Proceedings OOPSLA/ ECOOP ’90, vol. 25, no. 10, pp. 101-115, Oct 1990.

[22] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, New York, 1988.

[23] J.E.B. Moss and W.H. Kohler, “Concurrency Features for the Trellis/Owl Language,” Proceedings of ECOOP ‘87,BIGRE, no. 54, pp. 223-232, June 1987.

[24] O. Nierstrasz, “Active Objects in Hybrid,” Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 22, no. 12, pp. 243-253, Dec. 1987.

[25] O.M. Nierstrasz, “A Guide to Specifying Concurrent Behaviour with Abacus,” in *Object Management*, ed. D.C. Tsichritzis, pp. 267-293, Centre Universitaire d’Informatique, University of Geneva, July 1990, (submitted for publication).

[26] O.M. Nierstrasz and M. Papathomas, “Viewing Objects as Patterns of Communicating Agents,” ACM SIGPLAN Notices, Proceedings OOPSLA/ECOOP ’90, vol. 25, no. 10, pp. 38-43, Oct 1990.

[27] O.M. Nierstrasz and M. Papathomas, “Towards a Type Theory for Active Objects,” in *Object Management*, ed. D.C. Tsichritzis, pp. 295-304, Centre Universitaire d’Informatique, University of Geneva, July 1990.

[28] M. Papathomas, “Concurrency Issues in Object-Oriented Programming Languages,” in *Object Oriented Development*, ed. D.C. Tsichritzis, pp. 207-245, Centre Universitaire d’Informatique, University of Geneva, July 1989.

[29] M. Papathomas and D. Konstantas, “Integrating Concurrency and Object-Oriented Programming: An Evaluation of Hybrid,” in *Object Management*, Centre Universitaire d’Informatique, University of Geneva, ed. D. Tsichritzis, pp. 229-244, 1990.

[30] R.K. Raj and H.M. Levy, “A Compositional Model for Software Reuse,” Proceedings of the Third European Conference on Object-oriented Programming, pp. 3-24, Cambridge University Press, Nottingham, July 10-14, 1989.

[31] A. Snyder, “Encapsulation and Inheritance in Object-Oriented Programming Languages,” ACM SIGPLAN Notices, Proceedings OOPSLA ’86, vol. 21, no. 11, pp. 38-45, Nov 1986.

[32] C. Tomlinson and V. Singh, “Inheritance and Synchronization with Enabled Sets,” ACM SIGPLAN Notices, Proceedings OOPSLA ’89, vol. 24, no. 10, pp. 103-112, Oct 1989.

[33] A. Tripathi and M. Aksit, “Communication, Scheduling, and Resource Management in SINA,” JOOP, pp. 24-36, Nov/Dec 1988.

[34] J. Van Den Bos and C. Laffra, “PROCOL: A Parallel Object Language with Protocols,” ACM SIGPLAN Notices, Proceedings OOPSLA ’89, vol. 24, no. 10, pp. 95-102, Oct 1989.

[35] T. Watanabe and A. Yonezawa , “Reflection in an Object Oriented Concurrent Language,” SIGPLAN Notices, vol. 23, no. 11, pp. 306-315, ACM, 1988.

[36] P. Wegner, “Dimensions of Object-Based Language Design,” in *Proceedings OOPSLA ’87*, SIGPLAN Notices, vol. 22, pp. 168-182, ACM, Orlando, Florida, December 1987.

[37] A. Yonezawa, E. Shibayama, T. Takada and Y. Honda, “Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1,” in *Object-Oriented Concurrent Programming*, ed. M. Tokoro, pp. 55-89, The MIT Press, Cambridge, Massachusetts, 1987.

[38] Y. Yokote and M. Tokoro, “Concurrent Programming in ConcurrentSmalltalk,” in *Object-Oriented Concurrent Programming*, ed. M. Tokoro, pp. 129-158, The MIT press, Cambridge, Massachusetts, 1987.