A Classification of Various Approaches for Object-Based Parallel and Distributed Programming *

Jean-Pierre BRIOT

Dept. of Information Science
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku
Tokyo 113, Japan

fax: +81 (3) 5689-4365
briot@is.s.u-tokyo.ac.jp

Rachid GUERRAOUI

Département d'Informatique
École Polytechnique
Fédérale de Lausanne
CH-1015, Lausanne, Suisse

fax: +41 (21) 693-39-09
guerraoui@lse.epfl.ch

Abstract

This paper aims at classifying and discussing the various ways along which the "object" paradigm is used in parallel and distributed contexts. We distinguish the *applicative* approach, the *integrative* approach, and the *reflective* approach. The applicative approach applies object-oriented concepts, as they are, to structure parallel and distributed systems through libraries. The integrative approach consists in merging concepts such as: object and activity, message passing and transaction. The reflective approach integrates protocol libraries intimately within an object-based programming language. We discuss and illustrate each of these approaches and we point out their complementary levels and goals.

Keywords: Object, message passing, parallelism, distribution, transposition, application, libraries, integration, reflection, classification, combination.

*Tech Rep University of Tokyo, and École Polytechnique Fédérale de Lausanne, 1996.

1 Introduction

> *"Mr. President and fellow citizens of New York: The facts with which I shall deal this evening are mainly old and familiar; nor is there anything new in the general use I shall make of them. If there shall be any novelty, it will be in the mode of presenting the facts, and the inferences and observations following that presentation." Abraham Lincoln*[1]

It is now well accepted that the object paradigm provides good foundations for the new challenges of parallel, distributed and open computing. Object notions, and their underlying *message passing* metaphor, are strong enough to structure and encapsulate modules of computation, whereas the notions are flexible enough to match various granularities of software and hardware architectures.

Most of object-based programming languages do have some parallel or distributed extension(s), and almost every new architectural development in the distributed system community is, to some extent, object-based. For instance both the Open Distributed Processing (ODP) and the Object Management Group (OMG) recent standardization initiatives for heterogeneous distributed computing, are based on object concepts [NWM 93].

As a result, a lot of object-based parallel or distributed models, languages, or system architectures, have been proposed and described in the literature. Towards a better understanding and evaluation of these proposals, this paper discusses how object concepts are "articulated" (applied, customized, integrated, expanded...) with parallelism and distribution challenges and current technology[2]. Rather than an exhaustive study of various object-oriented parallel and distributed programming systems, this paper aims at classifying and discussing the various ways along which the object paradigm is used in parallel and distributed contexts.

Contribution of the Paper

The paper discusses and illustrates successively the three approaches we have identified: the *applicative* approach, the *integrative* approach and the *reflective* approach. The *applicative* approach applies object-oriented concepts, as they are, to structure parallel and distributed systems through libraries. The *integrative* approach consists in unifying parallel and distributed system concepts with object-oriented ones. The *reflective* approach integrates protocol libraries within an object-based programming language.

[1] Address at the Cooper Institute, New York, February 27, 1860.

[2] Various abstractions and mechanisms addressing parallelism and distribution issues have been proposed by various parallel programming, operating systems, distributed computing, and data-bases scientific communities. Some examples are: synchronization schemes, capabilities, and transactions.

Although these approaches may at first glance appear concurrent, we believe they are not. More precisely, the research directed along these approaches have complementary goals. The *applicative* approach is oriented towards system-builders and aims at identifying basic parallel and distributed abstractions. The *integrative* approach is oriented-towards application-builders, and aims at defining a high level programming language with few unified concepts. The *reflective* approach is oriented towards both application-builders and system-builders. The main goal is to provide the basic infrastructure to enable (dynamic) system customization with minimal impact on the application programs. The success of a reflective system relies both on a high level programming language, and on a rich library of parallel and distributed abstractions.

2 The Applicative Approach

2.1 Modularity and Structuring Needs

The basic idea of the *applicative* approach is to apply encapsulation and abstraction, and possibly also class and inheritance mechanisms, as a structuring tool to design and build parallel and distributed computing systems. In other words, the issue is in building and programming a parallel or distributed system, with an object-oriented methodology and programming language.

The main motivation is to increase modularity, by decomposing systems in various components with clear interfaces. This improves structuring of parallel and distributed systems, as opposed to Unix-style systems in which the different levels of abstraction are difficult to distinguish and thus to understand.

Applied to distributed operating systems, the *applicative* approach has led to a new generation of systems, such as Chorus [Roz92] and Choices [CIR+93], based on the concept of *micro-kernel*, and whose different services are performed by various specialized servers. Such systems are easier to understand, maintain and extend, and should also ultimately be more efficient as only the required modules have to be used for a given computation.

To illustrate the applicative approach, we survey the following examples: (1) the Smalltalk-80 programming language and environment, where a basic and simple object concept is uniformly applied to model and structure the whole system through class libraries, including concurrency and distribution aspects (2) the Eiffel programming language, for which various architectures of libraries have been proposed to address parallelism and distribution concerns, and (3) the Choices operating system, which organizes the architecture of a generic distributed operating system along abstract notions of class components, which may then be specialized for a given instantiation/porting of the (virtual) system.

2.2 Smalltalk

Smalltalk is often considered as one of the "purest" examples of object-oriented languages. This is because its "motto" is to have only a few concepts (object, message passing, class, inheritance) and to *apply* them *uniformly* to any aspect of the language and environment. One consequence is that the language is actually very *simple*. The richness of Smalltalk comes from its set of class *libraries*. They describe and implement various programming constructs (control structures, data structures…), internal resources (messages, processes, compiler…), and a sophisticated programming environment with integrated tools (browser, inspector, debugger…).

2.2.1 Standard Class Libraries.

Actually, even basic control structures, such as loop and conditional, are not primitive language constructs, but just standard methods of standard classes, which make use of the generic invocation of message passing. They are based on booleans and execution closures (*blocks*). Blocks, represented as instances of class `BlockContext`, are essential for building various control structures that the user may extend at his wish. They are also the basis for multi-threaded concurrency through *processes*. Standard class `Process` describes their representation and its associated methods implement process management (suspend, resume, adjust priority…). The behavior of the process scheduler is itself described by a class, named `ProcessorScheduler`.

The basic synchronization primitive is the semaphore, represented by class `Semaphore`. Standard libraries also include higher abstractions: class `SharedQueue` to manage communication between processes, and class `Promise` for representing the eager evaluation of a value computed by a concurrently executing process.

Smalltalk also offers libraries for remote communication, such as Sockets and RPC, as well as standard libraries for storage and exchange of object structures for persistence, transactions, and marshaling. The *Binary Object Streaming Service (BOSS)* library provides a basic support for building distribution mechanisms, e.g. persistence, transactions, and marshaling.

2.2.2 Extending Class Libraries.

Thanks to this uniform approach, concurrency concepts and mechanisms are well encapsulated and organized in a class hierarchy. Thus, they are much more understandable and extensible than if they were just simple primitives of a programming language. Furthermore, it is relatively easy to build up on the basic standard library of concurrency classes to construct more sophisticated abstractions. For concurrency abstractions, an example is the Simtalk platform [BÉZ87], which implements and classifies various synchronization and simulation abstractions (Hoare monitors, Kessels monitors, pessimistic or optimistic simulation objects...) upon Smalltalk standard abstractions/classes. For distributed and fault-tolerant programming abstractions, an example is the GARF project [GGM 94], in which two complementary class hierarchies have been developed for various communication models (point-to-point, multicast, atomic multicast...) and object models (monitor, persistent, replicated...). The HP Distributed Smalltalk product provides a set of distributed services following the OMG (CORBA) standard, themselves implemented as Smalltalk-80 class libraries.

In a similar approach, a library of classes (named "patterns" in Beta) for distributed programming has been developed with and for the Beta programming language [BM 94]. For instance, class `NameServer` represents a name server which maps textual object names to physical references. Class `ErrorHandler` manages partial errors/faults of a distributed system. The point in this work is to be able to add distributed features to a given sequential/centralized program, without changing the program logic, i.e. only through additions, as opposed to changes [BM 94, page 199].

2.3 Eiffel (Variations on)

Although the Eiffel programming language has been initially designed for sequential computation, several works have been conducted afterwards to address parallelism and distribution concerns. Some of them are described in [Mey 93a].

The approach proposed by Bertrand Meyer himself [Mey 93b] is applicative, in that it is an application of object concepts, and *minimalistic*, in that the idea is to expand the scope of existing standard Eiffel concepts and mechanisms (thus to increase their genericity) without introducing new ones, or rather the minimum. Notably, semantics of Eiffel assertions, by pre- and post-conditions, is redefined, when in a concurrent context, as waiting until conditions are satisfied[3].

Additional constructions and mechanisms being often necessary, they may be described and implemented through libraries, by applying the object methodology to organize them. For instance, class `Concurrency` [KB 93] encapsulates an activity (associated to an object) and remote asynchronous message passing.

Alternatively, but following the same applicative approach, parallelism and distribution have been introduced in the ÉPÉE environment [Jéz 93] at the level of complex data structures[4] (e.g. a matrix). ÉPÉE provides libraries of abstract structures which may be distributed among several processors, without any addition to the Eiffel language.

[3] along the principles of behavioral synchronization, later detailed in Sect. 3.4.2.

[4] ÉPÉE follows a data-parallelism (SPMD) model of parallelism, as opposed to an activation/control-parallelism (MIMD) model, as for class `Concurrency` (and, more generally speaking, for integrative languages based on the concept of *active objet*, as we will later see in Sect. 3.3). Indeed, objects represent duality (and unification) between data and procedures (potential activation).

2.4 Choices

Choices [CIR+93] is a *generic* operating system developed at the University of Illinois at Urbana-Champaign since 1987. The objective is not only to be easily ported onto various machines, but also to be able to adjust various characteristics of both hardware, resources, and application interfaces such as: file format, communication network, and memory model (shared or distributed). An object-oriented methodology is presented together with the system, both for the design of distributed applications, and for the design of new extensions to the Choices kernel.

A specific C++ class library has been developed. For instance, class `ObjectProxy` implements remote communications between objects, classes `MemoryObject` and `FileStream` represent memory management, and class `ObjectStar` provides some generalized notion of pointer. Class `ObjectStar` provides transparency for remote communications without need for a precompilation step. This class is also useful for the automatic garbage collector. Class `Disk` abstracts and encapsulates a physical storage device which may be instantiated, e.g. in class `SPARCstationDisk` when porting Choices onto a SPARC station.

The experience of the Choices projects shows that a distributed operating system, developed with an object-oriented methodology and programming language, helps at achieving better genericity and extensibility.

2.5 In Search for Standard Abstractions

More generally speaking, the main issue underlying the applicative approach is the design and implementation of adequate abstractions on top of which various parallel and distributed schemes can be built.

One of the most significant example for concurrent programming is the *semaphore* abstraction which, through a well defined interface (`wait` and `signal` operations), and a known behavior (metaphor of the train semaphores), represents one standard of synchronization for concurrent programming. Such a basic abstraction may then be used as a foundation to build various higher-level synchronization mechanisms. Classification and specialization mechanisms, as offered by object-oriented programming, are then appropriate to organize such a library/hierarchy of abstractions, as for instance in the the Simtalk platform (Sect. 2.2).

An example of development of concurrency abstractions, complementary to program abstractions, may be found with the Demeter project. In the Demeter environment, the abstract specification of a program is decomposed into two loosely coupled dimensions: the "structural block", which represents relations between classes, and the "behavioral block", which describes the operations. A third dimension has been recently added: the "concurrency block", which describes the abstract synchronization patterns between processes [LL94]. The abstract specifications and the relative independence between these three components help at the development of generic and reusable programs.

Andrew Black has proposed a similar study of abstractions for distributed programming [BLA 91]. He suggested, as a first exercise, to decompose the concept of transaction into a set of abstractions. The goal is to represent concepts such as *lock*, *recovery*, and *persistence*, through a set of objects that must be provided by a system in order to support transactions. The modularity of this approach would help defining various transaction models, adapted to specific kinds of applications. For instance, a computer supported cooperative application does not need strong concurrency control constraints as for a banking application[5]. Both Venari [WIN 94] and Phoenix [GS 95] projects aim at defining various transactional models from a set of minimal abstractions.

2.6 Evaluation of the Applicative Approach

In summary, the applicative approach aims at increasing the flexibility, yet reducing the complexity, of parallel and distributed computing systems by structuring them as libraries of classes. Each aspect or service is represented by an object. Such modularity and abstraction objectives are very important because parallel and distributed computing systems are complex systems, which ultimately use very low-level mechanisms, e.g. network communication. Furthermore, such systems are often developed by teams of programmers, and in such context, having separate modules with well defined interfaces is of primary importance. Finally, the difficulty with maintaining and extending Unix-like systems comes mainly from their low modularity and insufficient level of abstraction.

Although progress is made towards that direction, as noted above, it is still too early to exhibit a standard class library for parallel and distributed programming. We need, both a good knowledge of the minimal mechanisms required, and also a consensus on a set of such mechanisms, involving different technical communities, notably: programming languages, operating systems, distributed systems, and data-bases. The fact that the semaphore abstraction became a standard primitive for synchronization, leads us to think that other abstractions for parallel and distributed programming could also be identified and adopted.

3 The Integrative Approach

3.1 Unification Needs

The amount of issues and concepts required represents one of the major difficulties of parallel and distributed programming. In addition to classical constructs of sequential programming, parallel and distributed computation introduces concepts such as *process*, *semaphore*, *monitor* and *transaction*. The applicative approach helps at structuring parallelism and distribution concepts and mechanisms, but it keeps them disjoint from

[5]The second kind of application requires a strict serialization of transactions through a locking mechanism, whereas the first kind does not need it.

the objects structuring the application programs. In other words, the programmer still faces at least two major different issues: programming with objects, *and* managing parallelism and distribution of the program, also with objects, but *not the same objects !*

Furthermore, when using libraries, the programming style may become a little cumbersome, as the distribution aspects (and more specifically the manipulation of the objects implementing them) add up to the standard programming style. For instance, the introduction of asynchronous and remote communication model in Eiffel class `Concurrency` (see Sect 2.3), forces to some amount of explicit message manipulation (e.g. see [KB 93, pages 109–11]), as opposed to standard implicit message passing. One may then choose to *integrate* such a construct directly into the programming language as a language extension, such as Eiffel// [Car 93], or a brand new language.

In summary, rather than leaving both the object programs and the management of parallelism and distribution orthogonal, the *integrative* approach aims at merging them by integrating concepts, and offering a unified object model for the programmer.

3.2 Levels of Integration

There are various possible levels of integration between object-oriented programming concepts and parallelism and distribution concepts. We will consider three main different levels. Note that they are relatively independent of each other. Thus, as we will see, a given language or system may follow one level of integration but not another one.

A first integration between the concept of an object and the concept of a process (more generally speaking the concept of an autonomous activity) leads to the concept of an *active object*. Indeed, an object and a process may both be considered as communicating encapsulated units[6]. Actor languages [Lie 87, Agh 86] are a typical example of programming languages based on the notion of an active object.

A second level of integration associates synchronization to object activation, leading to the notion of a *synchronized object*. Message passing is then considered as an implicit synchronization between the sender and the receiver. Furthermore one often associates mechanisms for controlling the activation of invocations at the level of an object, e.g. by attaching a guard to each method. Note that the concept of an active object already implies some form of synchronized object, as the existence of a (single) activity private to the object actually enforces the serialization of invocations. However, some languages or systems, e.g. Guide [BLR 91] or Arjuna [PS 88], associate synchronization to objects although they distinguish the notions of object and activity.

A third level of integration considers the object as the unit of distribution, leading to the notion of a *distributed object*. Objects are seen as entities which may be distributed and replicated on several processors.

[6] This similarity has been for instance noted in [Mey 93b].

The message passing metaphor is seen as a transparent way of invoking either local or remote objects. The Emerald distributed programming language [BHJ+86] is an example of distributed programming language based on the notion of distributed object. One can also further integrate message passing with the transaction concept as to support inter-object synchronization and fault-tolerance [LS 83, GCL+92].

3.3 Active Objects

The basic idea leading to the concept of an *active object* is to consider an object having its own computing resource, i.e. its own private activity. This approach, simple and natural, is quite influent [YT 87], following the way traced by actor-languages [Lie 87, Agh 86]. The concept of an active object is also a natural foundation for building higher-level autonomous *agents*, for distributed knowledge-based systems.

The independence of object activities provides what is usually called *inter-object* concurrency. When, for each active object, requests are usually processed one at a time: this is called a *serialized object*. In other computation models (e.g. Actors [Agh 86]) an active object is allowed to process several requests simultaneously, thus owning more than one internal activity: this is called *intra-object* concurrency. This increases the expressive power as well as the overall concurrency. But this requires a further concurrency control in order to ensure object state consistency (as we will see below).

3.4 Synchronized Objects

The presence of concurrent activities requires some degree of synchronization, i.e. constraints, in order to ensure a correct execution of programs. Synchronization may be associated to objects and to their communication means, i.e. message passing, through various (sub-)levels of identification.

3.4.1 Synchronization at Message Passing Level.

A straightforward transposition of the message passing mechanism, from a sequential computing context to a concurrent one, leads to the implicit synchronization of the sender (caller) to the receiver (callee). This is called *synchronous* transmission: to resume its own execution, the sender object waits for (1) completion by the receiver of the invoked method execution and then (2) the return of the reply.

In case of active objects, the sender and the receiver own independent activities. It is therefore useful to introduce some *asynchronous* type of transmission, where the sender resumes its execution as soon after sending the message, i.e. without waiting for completion of the invoked method by the receiver. This type of transmission introduces further concurrency through communication. It is well suited for a distributed architecture, because if the receiver (the server) is located on a distant processor, the addition of the communication latency to the processing time may be significant. Finally, some languages (e.g. see in [YT 87]) introduce some mixed kind of transmission, which immediately returns an eager promise for (handle to) a (future) reply, without waiting for the actual completion of the invocation.

3.4.2 Synchronization at Object(s) Level.

The identification of synchronization with message passing, that is with requests invocation, has the advantage of transparently ensuring some significant part of the synchronization concerns. Indeed synchronization of requests is transparent to the client object, being managed by the object serving requests.

In case of serialized active objects, requests are processed one at a time, according to their order of arrival. Some finer grain or rather more global concurrency control may however be necessary for objects. We will then consider three different levels of synchronization at the object(s) level. They respectively correspond to: the internal processing of an object, its interface, and the coordination between several objects.

Intra-Object Synchronization. In case of *intra-object* concurrency (i.e. an object computing simultaneously several requests), it is necessary to include some concurrency control in order to ensure the consistency of the object state. Usually, the control is expressed in terms of exclusions between operations[7]. The typical example is the readers and writers problem, where several readers are free to access simultaneously to a shared book. However, the presence of one writer excludes all others (writers and readers).

Behavioral Synchronization. It is possible that an object may temporarily not be able to process a certain kind of request which is nevertheless part of its interface. The typical example is the bounded buffer example, which may not accept some insertion request while it is full. Rather than signaling an error, it may delay the acceptance of that request until it is no more full. This makes synchronization of services between objects being fully transparent.

Inter-Objects Synchronization. Finally, it may be necessary to ensure some consistency, not just individual, but also global (coordination) between mutually interacting objects. Consider the example of a money transfer between two bank accounts. The issue is in ensuring the invisibility of possible transient and inconsistent global states, while the transfer takes place. Intra-object or behavioral synchronization are not sufficient. We must introduce a notion such as an atomic transaction [BHG 87], to coordinate the different invocations.

[7] Note that the case of a mutual exclusion between all methods subsumes the case of a serialized object (as defined in Sect. 3.3).

3.4.3 Synchronization Schemes.

Various *synchronization schemes* have been proposed to address these various levels of concurrency control. Many of them are actual derivations from general concurrent programming and have been more and less integrated within the object-oriented concurrent programming framework.

Centralized schemes, as for instance *path expressions*, specify in an abstract way the possible interleavings of invocations, and may be associated in a natural way to a class. The Procol language [vdBL 91] is based on that idea. Another example of centralized scheme is the concept of *body*. That is, some distinguished centralized operation (the *body*), explicitly describes the types and sequence of requests that the object will accept during its activity[8]. Languages like POOL [Ame 87] and Eiffel// [Car 93] are based on this concept.

Decentralized schemes, such as *guards*, are based on boolean activation conditions, that may be associated to each method. Synchronization counters are counters recording the invocation status for each method, i.e. the number of received, started and completed invocations. Associated to guards, they provide a very fine grained control of intra-object synchronization. An example is the distributed programming language Guide [BLR 91].

Finally, a higher level formalism is based on the notion of *abstract behaviors*. This scheme is quite appropriate for behavioral synchronization (introduced in the previous section). The idea[9] is as following: an object conforms to some abstract behavior representing a set of *enabled methods*. In the example of the bounded buffer, three abstract behaviors are needed: `empty`, `full`, and `partial`. The abstract behavior `partial` is expressed as the union of `empty` and `full`, and consequently is the only one to enable both insertion and extraction methods. After completing the processing of an invocation, next abstract state is computed to possibly update the state and services availability of the object.

Note that, although integration of synchronization schemes with object model is usually straightforward, this integration impacts on the reuse of synchronization specifications (see Sect. 3.6.1).

3.5 Distributed Objects

An object represents an independent unit of execution, encapsulating data, procedures, and possibly private resources (activity) for processing the requests. Therefore a natural option is to consider an object as the unit of distribution, and possible replication. Furthermore, self-containedness of objects (data plus procedures, plus possible internal activity) eases the

[8] This concept is actually a direct offspring of Simula-67 [BDM+ 73] concept of *body*, which actually included support for coroutines. Note that this initial potential of objects for concurrency was then abandoned, both for technological and cultural reasons, by most of Simula-67 followers. This potentiality started being "rediscovered" from the late 70's, actor-languages appearing as new pioneers [Lie 87].

[9] See e.g. [MY 93] for a more detailed description.

issue of moving and migrating them around. Also, note that message passing not only ensures the separation between services offered by an object and its internal representation, but also provides the independence of its physical location. Thus, message passing may subsume both local and remote invocation (whether sender and receiver are on the same or distinct processors is transparent to the programmer) as well as possible unaccessibility of an object/service.

3.5.1 Accessibility and Fault-Tolerance.

In order to handle unaccessibility of objects, in the Argus distributed operating system [LS 83], the programmer may associate an exception to an invocation. If an object is located on a processor which is unaccessible, because of a network or processor fault, an exception is raised, e.g. to invoke another object. A transaction is implicitly associated to each invocation (synchronous invocation in Argus), to ensure atomicity properties. For instance, if the invocation fails (e.g. if the server object becomes unaccessible), the effects of the invocation are canceled. The Karos distributed programming language [GCL+92] extends the Argus approach by allowing the association of transactions also to asynchronous invocations.

3.5.2 Migration.

In order to improve the accessibility of objects, some languages or systems support mechanisms for object migration. In the Emerald distributed programming language [BHJ+86], and the COOL generic run-time layer [LJP 93], the programmer may decide to migrate an object from one processor to another. He may control (in terms of *attachements* in Emerald) which other related objects should also migrate together.

3.5.3 Replication.

As for migration, a first motivation of replication is in increasing the accessibility of an object, by replicating it onto the processors of its (remote) clients. A second motivation is fault-tolerance. By replicating an object on several processors, its services become robust against possible processor failure. In both cases, a fundamental issue is to maintain the consistency of the replicas, i.e. to ensure that all replicas hold the same values. In the Electra [Maf 95] distributed system, the concept of remote invocation has been extended in the following fashion: invoking an object leads to the invocation of all its replicas while ensuring that concurrent invocations are ordered along the same (total) order for all replicas. Andrew Black also introduced a general mechanism for group invocation well suited for replicated objects [BI 93].

3.6 Limitations

The integrative approach attempts at unifying object mechanisms with parallelism and distribution mechanisms. Meanwhile, some conflicts may arise between them, as we will see below.

3.6.1 Inheritance Anomaly.

Inheritance mechanism is one of the key mechanisms for achieving good reuse of object-oriented programs. It is therefore natural to use inheritance to specialize synchronization specifications associated to a class of objects. Unfortunately, experience shows that: (1) synchronization is difficult to specify and moreover to reuse, because of the high interdependency between the synchronization conditions for different methods, (2) various uses of inheritance (to inherit variables, methods, and synchronizations), may conflict with each other, as noted in [McH 94, BOM 95]. In some cases, defining a new subclass, even only with one additional method, may force the redefinition of all synchronization specifications. This limitation has been named the *inheritance anomaly phenomenon* [MY 93].

Specifications along centralized schemes (see Sect. 3.4.3) turn out to be very difficult to reuse, and often must be completely redefined. Decentralized schemes, being modular by essence, are better suited for selective specialization. However, this fine-grained decomposition, down at the level of each method, partially backfires. This is because synchronization specifications, even if decomposed for each method, still remain more or less interdependent. As for instance in the case of intra-object synchronization with synchronization counters, adding a new write method in a subclass may force redefinition of other methods guards, in order to account for the new mutual exclusion constraint. (See [MY 93] for a detailed analysis and classification of the possible problems.)

Among the recent directions proposed for minimizing the problem, we may cite: (1) specifying and specializing independently behavioral and intra-object synchronizations [Tho 92], (2) allowing the programmer to select among several schemes [MY 93], and (3) genericity, by instantiating abstract specifications, as an alternative to inheritance for reusing synchronization specifications [McH 94].

3.6.2 Compatibility of Transaction Protocols.

It is tempting to integrate transaction concurrency control protocols into objects. Thus one may locally define, for a given object, the optimal concurrency control or recovery protocol. For instance, commutativity of operations enables the interleaving (without blocking) of transactions on a given object. Unfortunately, the gain in modularity and specialization may lead to incompatibility problems [Wei 89]. Broadly speaking, if objects use different transaction serialization protocols (i.e. serialize the transactions along different orders), global executions of transactions may become inconsistent, i.e, non serializable. A proposed approach to handle that problem is in defining local conditions, to be verified by objects, in order to ensure their compatibility [WEI 89, GUE 95b].

3.6.3 Replication of Objects and Communications.

The communication protocols which have been designed for fault-tolerant distributed computing (see Sect. 3.5.3) consider a standard client/server model. The straightforward transposition of such protocols to the object model leads to the problem of unexpected duplication of invocations. Indeed, an object usually acts conversely as a client and as a server. Thus an object which has been replicated as a server may itself in turn invoke other objects (as a client). As a result all replicas of the object will invoke these other objects several times. This unexpected duplication of invocations may lead, in the best case, to inefficiency, and in the worst case, to inconsistencies (by invoking several times the same operation). A solution, proposed in [MGG 95], is based on *pre-filtering* and *post-filtering*. Pre-filtering consists in coordinating processing by the replicas (when considered as a client) in order to generate a single invocation. Post-filtering is the dual operation for the replicas (when considered as a server) in order to discard redundant invocations.

3.6.4 Factorization vs Distribution.

Last, a more general limitation (i.e. less specific to the integrative approach) comes from standard implementation frameworks for object factorization mechanisms, which usually rely on strong assumptions about centralized (single memory) architectures.

The concept of class variables[10], supported by several object-oriented programming languages, is difficult and expensive to implement for a distributed system. Unless introducing complex and costly transaction mechanisms, their consistency is hard to maintain, once instances of a same class may be distributed among processors. Note that this problem is general for any kind of shared variable. Standard object-oriented methodology tends to forbid the use of shared variables, but may advocate using class variables instead.

In a related problem, implementing inheritance on a distributed system leads to the problem of accessing remote code for superclasses, unless all class code is replicated to all processors, which has obvious scalability limitations. A semi-automatic approach consists in grouping classes into autonomous modules as to help at partitioning the class code among processors.

Some radical approach replaces the inheritance mechanism between classes, by the concept/mechanism of *delegation* between objects. This mechanism has actually been introduced in the Actor concurrent programming language Act 1 [LIE 87]. Intuitively, an object which may not

[10] As supported by Smalltalk.

understand a message will then delegate it (i.e. forward it[11]) to another object, called its *proxy*. The proxy will process the message in place of the initial receiver, or it can also itself delegate it further to its own designated proxy. This alternative to inheritance is very appealing as it only relies on message passing, thus is fits well with a distributed implementation. Meanwhile, the delegation mechanism needs some non trivial synchronization mechanism to ensure the proper handling (ordering) of recursive messages, prior to other incoming messages. Thus, it may not offer a general and complete alternative solution [BY 87].

3.7 Evaluation of the Integrative Approach

In summary, the integrative approach is very appealing by the merging it achieves between concepts, from object-oriented programming, and those from parallel and distributed programming. It thus provides a minimal number of concepts and a single conceptual framework to the programmer. Nevertheless, as we discussed in Sect. 3.6, this approach unfortunately suffers from limitations in some aspects of the integration.

Another potential weakness is that some too systematic unification/integration may lead to some too restrictive model ("too much uniformity kills variety !") and may lead to inefficiencies. For instance, stating that every object is active, and/or every message transmission is a transaction, may be inappropriate for some applications not necessarily requiring such protocols, and their associated computational load. A last important limitation is a *legacy* problem, that is the possible difficulty with reusing standard sequential program. Some straightforward way is the encapsulation of sequential programs into active objects. But, note that a cohabitation between active objects and standard ones, called *passive objects*, is non homogeneous, which requires methodological rules for distinction between active objects and passive objects [CAR 93].

4 The Reflective Approach

As we earlier discussed, the applicative approach (library-based approach) helps at structuring parallel and distributed programming concepts and mechanisms, thanks to encapsulation, genericity, class, and inheritance concepts. The integrative approach minimizes the amount of concepts to be mastered by the programmer and makes mechanisms more transparent, but at the cost of possibly reducing the flexibility and the efficiency of mechanisms offered. Indeed programming languages or systems built from libraries are often more extensible than languages designed along the integrative approach. Libraries help at structuring and simulating various solutions, and thus usually bring good flexibility, whereas brand new languages may freeze too early their computation and communication

[11]Note that, in order to handle recursion properly, the delegated message will include the initial receiver.

models. In other words it would be interesting to keep the unification and simplification advantages of the integrative approach, while retaining the flexibility of the applicative/library approach.

One important observation is that the applicative approach and the integrative approach actually address *different* levels of concerns and use: the integrated approach is for the application programmer, and the applicative approach is for the system programmer. In other words, the end user programs its applications, with an integrative (simple and unified) approach in mind. The system programmer, or the more expert user, builds or customizes the system, through the design of libraries of protocol components, along an *applicative* approach.

Therefore, and as opposed to what one may think at first, the *applicative* approach and the *integrative* approach are *not* in competition, but rather *complementary*. The issue is then: "How can we actually combine these two levels of programming ?", and to be more precise: "How do we *interface them* ?". It turns out that a general methodology for adapting the behavior of computing systems, named *reflection*, offers such kind of a "glue".

4.1 Reflection

Reflection is a general methodology to describe, control, and adapt the behavior of a computational system. The basic idea is to provide a representation of the important characteristics/parameters of the system in terms of the system itself. In other words, (static) representation characteristics, as well as (dynamic) execution characteristics, of application programs are made concrete into one (or more) program(s), which represents the default computational behavior (interpreter, compiler, execution monitor...). Such a description/control program is called a *meta-program*. Specializing such programs enables to customize the execution of the application program, by possibly changing data representation, execution strategies, mechanisms and protocols. Note that the *same* language is used, both for writing application programs, *and* for meta-programs controlling their execution. However, the complete separation between the application program and the corresponding meta-programs is strictly enforced.

Reflection helps at decorrelating libraries specifying implementation and execution models (execution strategies, concurrency control, object distribution) from the application program. This increases modularity, readability and reusability of programs. Last, reflection provides a methodology to open up and make adaptable, through a *meta-interface*[12], implementation decisions and resources management, which are often hardwired and fixed, or delegated by the programming language to the under-

[12] This *meta-interface* enables the client programmer to adapt and tune the *behavior* of a software module, independently of its *functionalities*, which are accessed through the standard (*base*) interface. This has been named by Gregor Kiczales the concept of *open implementation* [Kic 94].

lying operating system.

In summary, reflection helps at integrating protocol libraries intimately within a programming language or system, thus providing the interfacing framework (the "glue") between the applicative and the integrative approaches/levels.

4.2 Reflection and Objects

Reflection fits specially well with object concepts, which enforce a good encapsulation of levels and a modularity of effects. It is therefore natural to organize the control of the behavior of an object-oriented computational system (its meta-interface) through a set of objects. This organization is named a *Meta-Object Protocol (MOP)* [KdR+91], and its components are called *meta-objects* [Mae87], as meta-programs are represented by objects. They may represent various characteristics of the execution context such as: representation, implementation, execution, communication and location. Specializing meta-objects may extend and modify, locally, the execution context of some specific objects of the application program.

Reflection may also help at expressing and controlling resources management, not only at the level of an individual object, but also at a broader level such as: scheduler, processor, name space, object group..., such resources being also represented by meta-objects. This helps at a very fine-grained control (e.g. for scheduling and load balancing) with the whole expressive power of a full programming language [OI94], as opposed to some global and fixed algorithm (which is usually optimized for a specific kind of application or an average case).

4.3 Examples of Meta Object Protocols (MOPs)

The CodA architecture [McA95] is a representative example of a general object-based reflective architecture (i.e. a "MOP") based on *meta-components*[13]. CodA considers by default seven (7) meta-components, associated to each object, corresponding to: *message sending*, *receiving*, *buffering*, *selection*, *method lookup*, *execution*, and *state accessing*. An object with default meta-components behaves as a standard (sequential and passive) object[14]. Attaching specific (specialized) meta-components allows to selectively changing a specific aspect of the representation or execution model for a single object. A standard interface between meta-components helps at composing meta-components from different origins.

Note that some other reflective architectures may be more specialized and may offer a more reduced (and abstract) set of meta-components. Examples are the Actalk and GARF platforms, where a smaller amount of

[13] Note that meta-components are indeed meta-objects. In the following, we will rather use the term *meta-component* in order to emphasize the pluggability aspects of a reflective architecture (MOP) such as CodA. Also, for simplification, we will often use the term *component* in place of *meta-component*.

[14] to be more precised, as a standard Smalltalk object, as CodA is currently implemented in Smalltalk.

meta-components may be in practice sufficient to express a large variety of schemes and application problems. The Actalk platform [Bri 89, Bri 96] helps at experimenting with various synchronization and communication models for a given program, by changing and specializing various models/components of: (1) *activity* (implicit or explicit acceptance of requests, intra-objet concurrency...) and *synchronization* (abstract behaviors, guards...), (2) *communication* (synchronous, asynchronous...), and (3) *invocation* (time stamp, priority...). The GARF platform [GGM 94], for distributed and fault-tolerant programming, offers a variety of mechanisms along two dimensions/components: (1) object control (persistence, replication...) and (2) communication (multicast, atomic...).

More generally speaking, depending on the actual goals and the balance expected between flexibility, generality, simplicity and efficiency, design decisions will dictate the amount and the scope of the mechanisms which will be "opened-up" to the meta-level. Therefore, some mechanisms may be represented as *reflective methods*, but which belong to *standard* object classes, that is, without explicit and complete meta-objects.

Smalltalk is a representative example of that latter category. In addition to the (meta-)representation of the program structures and mechanisms, as first class objects (see Sect. 2.2), a few very powerful reflective mechanisms offer some control over program execution. Examples are: redefinition of error handling message, reference to current context, references swap, changing the class of an object... Such facilities actually help at easily building and integrating various platforms for concurrent, parallel and distributed programming, such as Simtalk, Actalk, GARF, and CodA itself.

4.4 Examples of Applications

To illustrate how reflection may help at mapping various computation models and protocols onto user programs, we will quickly survey some examples of experiments with a specific reflective architecture. (We chose CodA. See [McA 95] for a more detailed description of its architecture and libraries of components.)

Note that, in the case of the CodA system, as well as for almost all other examples of reflective systems further described, the basic programming model is *integrative*, while reflection enables the customization of parallelism and distribution aspects and protocols, by specializing *libraries* of meta-components.

4.4.1 Concurrency Models.

In order to introduce concurrency for a given object (by making it into an *active* object, along an integrated approach), two meta-components are specialized. The specialized *message buffering* component[15] is a queue

[15] The default *buffering* component is actually directly passing incoming messages on to the *execution* component.

which will buffer incoming messages. The specialized *execution* component associates an independent activity (thread) to the object. This thread processes an endless loop for selecting and performing next message from the *buffering* component.

4.4.2 Distribution Models.

In order to introduce distribution, a new meta-component is *added*, for *marshaling* messages to be remotely sent. In addition, two new specific objects are introduced, which represent the notion of a *remote reference* (to a remote object) and the notion of a (memory/name) *space*. The remote reference object has a specialized *message receiving* component, which marshals the message into a stream of bytes and sends it through the network to the actual remote object. This latter one has another specialized *message receiving* component which reconstructs and actually receives the message. Marshaling decisions, e.g. which argument should be passed by reference, by value (i.e. a copy), up to which level..., may be specialized by a *marshaling descriptor* supplied by the *marshaling* component.

4.4.3 Migration and Replication Models.

Migration is introduced by a new meta-component which describes the form and the policies (i.e. when it should occur) for migration. Replication is managed by adding two new dual meta-components. The first one is in charge of controlling access to the state of the original object. The second one controls the access to each of its replicas. Again, marshaling decisions, such as: which argument should be passed by reference, by value, by move (i.e. migrated, as in Emerald [BHJ+86]), with attachments..., may be specialized through the *marshaling descriptors* supplied by the corresponding component. Also one may specialize aspects such as which parts of the object should be replicated, and various management policies for enforcing the consistency between the original object and its replicas.

4.5 Other Examples of Reflective Architectures

Other examples of representative reflective architectures and their applications are quickly mentioned in the following. Note that this is by no means an exhaustive study.

4.5.1 Dynamic Installation and Composition of Protocols.

The general MAUD methodology [AFP+93] focuses on fault tolerance protocols, such as: server replication, check point... Its specificity is in offering a framework for *dynamic installation* and *composition* of specialized meta-components. The dynamic installation of meta-components enables the installation of a given protocol only when needed, and without stopping the program execution. The possibility to associate meta-components, not only to objects, but also to other meta-components (which are first-class objects), enables the layered composition of protocols.

4.5.2 Control of Migration.

The autonomy and self-containedness of objects, further reinforced in the case of active objects, makes them easier to migrate "as a single piece". Nevertheless, the decision to migrate an object is an important issue which often remains with the programmer responsibility (e.g. in Emerald [BHJ+86]). It may be interesting to semi-automate such a decision, along various considerations such as: processor load, ratio of remote communications... Reflection helps at integrating such statistical data (residing for physical and shared resources), and at using them by various migration algorithms described at the meta-level [OI94].

4.5.3 Customizing System Policies.

The Apertos distributed operating system [Yok92], represents a significant and innovative example of a distributed operating system, completely designed along an object-based reflective architecture (MOP). In supplement to the modularity and the genericity of the architecture gained by using an applicative (object-oriented) approach (as for Choices, already discussed in Sect. 2.4), reflection brings the (possibly dynamic) customization of the system towards application requirements. As for instance one may easily specialize the scheduling policy in order to support various kinds of schedulers, e.g. a real-time scheduler. Another gain is in the size of the micro-kernel obtained, which is particularly small, as it is reduced to supporting the basic reflective operations and the basic resources abstractions. This helps at both the understanding and the porting of the system.

4.5.4 Reflective Extension of an Existing Commercial System.

A reflective methodology has recently been used in order to incorporate extended[16] transaction models into an *existing* commercial transaction processing system. It extends a standard transaction processing monitor, in a minimal and disciplined way (based on "upcalls"), to expose features such as: lock delegation, dependency tracking between transactions, and definition of conflicts, and to represent them as reflective operations [BP95]. These reflective primitives are then used to implement various extended transaction model, such as: split/join, cooperative groups...

[16] That is, relaxing some of the standard (ACID) transaction properties.

4.6 Related Frameworks for Interfacing Customizations.

We finally mention two examples of frameworks for customizing computational behaviors, which are closely related to reflection.

4.6.1 The Composition-Filters Model.

The SINA language is based on the notion of a *filter*, a way to specify arbitrary manipulation and actions for messages sent to (or from) an object [AWB+94]. In other words, filters represent some reification of the communication and interpretation mechanism between objects. By combining various filters for a given object, one may construct complex interaction mechanisms in a composable way.

4.6.2 Generic Run-Time as a Dual Approach.

The frontier between programming languages and operating systems is getting thinner. Reflective programming languages have some high-level representation of the underlying execution model. Conversely, and dual to reflection, several distributed operating systems provide a generic run time layer, as for instance the COOL layer for the Chorus operating system [LJP93]. These generic run time layers are designed as to be used by various programming languages, thanks to some "upcalls" which delegate specific representation decisions to the programming language.

4.7 Evaluation of the Reflective Approach

Reflection provides a general framework for the customization of parallelism and distribution aspects and protocols, by specializing and *integrating* (meta)-*libraries* intimately within a language or system, while *separating* them from the application program.

Many reflective architectures are currently proposed and getting evaluated. It is too early yet to find and validate some general and optimal reflective architecture for parallel and distributed programming (although we believe that CodA [McA95] is a promising step in that direction). Meanwhile, we now need more experience in the practical use of reflection, to be able to find good tradeoffs between the flexibility required, the architecture complexity, and the resulting efficiency. One possible (and currently justified) complain is about the actual relative complexity of reflective architectures. Nevertheless, and independently of the required cultural change, we believe that this is the price to pay for the increased, albeit disciplined, flexibility that they offer. Another significant current limitation concerns efficiency, consequence of extra indirections and interpretations. Partial evaluation (also called program specialization) is currently proposed as a promising technique to minimize such overheads [MMA+95].

5 Conclusion

Towards a better understanding and evaluation of various object-based parallel and distributed developments, we have proposed a classification of the different ways along which the object paradigm is used in parallel and distributed contexts. We have identified three different approaches which convey different, yet complementary, research streams in the object-based parallel and distributed system community.

The *applicative* approach (library-based approach) helps at structuring parallel and distributed programming concepts and mechanisms, through encapsulation, genericity, class, and inheritance concepts. The principal limitation of the approach is that the programming of the application, and of the parallel and distribution architecture, are represented by unrelated sets of concepts and objects. The applicative approach can be viewed as a bottom-up approach and is directed towards system-builders.

The *integrative* approach minimizes the amount of concepts to be mastered by the programmer and makes mechanisms more transparent, by providing some unified parallel and distributed high level object model. However, this is at the cost of possibly reducing the flexibility and efficiency of the mechanisms. The integrative approach can be viewed as a top-down approach and is directed towards application-builders.

Finally, by providing a framework for integrating protocol libraries intimately within a programming language or system, the *reflective* approach provides the interfacing framework (the "glue") between the applicative and the integrative approaches/levels. Meanwhile, it enforces the separation of their respective levels. In other words, reflection provides the *meta-interface* through which the system designer may install system customizations and thus change the execution context (parallel, distributed, fault tolerant, real time, adaptive...) with minimal changes on the application programs.

The reflective approach also contributes in blurring the distinction between programming language, operating system, and data base, and at easing the development, adaptation and optimization of a minimal computing system dynamically extensible. Nevertheless, we should strongly remind that this does not free us from the necessity of a good basic design and finding a good set of foundational abstractions.

References

[Agh 86] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, Series in Artificial Intelligence, MIT Press, 1986.

[AFP+93] G. Agha, S. Frølund, R. Panwar, and D. Sturman, "A Linguistic Framework for Dynamic Composition of Dependability Protocols," *Dependable Computing for Critical Applications III (DCCA-3)*, IFIP Transactions, Elsevier, 1993, pages 197–207.

[AWB+94] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa, "Abstracting Object Interactions Using Composition Filters," [GNR94], pages 152–184.

[Ame87] P. America, "POOL-T: A Parallel Object-Oriented Language," [YT87], pages 199–220.

[BLR91] R. Balter, S. Lacourte, and M. Riveill, "The Guide Language," *The Computer Journal*, Special Issue on Distributed Operating Systems, Vol. 37, n° 6, CEPIS - Oxford University Press, 1994, pages 519–530.

[BOM95] C. Baquero, R. Oliveira and F. Moura, "Integration of Concurrency Control in a Language with Subtyping and Subclassing," *USENIX COOTS Conference (COOTS'95)*, Monterey, CA, USA, June, 1995.

[BP95] R. Barga and C. Pu, "A Practical and Modular Implementation of Extended Transaction Models," Technical Report, n° 95-004, CSE, Oregon Graduate Institute of Science & Technology, Portland OR, USA, 1995.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[Béz87] J. Bézivin, "Some Experiments in Object-Oriented Simulation," *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, Vol. 22, n° 12, December 1987, pages 394–405.

[BDM+73] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *Simula Begin*, Petrocelli Charter, 1973.

[BHJ+86] A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, "Distribution and abstract types in Emerald," *IEEE Transactions on Software Engineering*, Vol. 12, n° 12, December 1986.

[Bla91] A. Black, "Understanding Transactions in the Operating System Context," *Operating Systems Review*, Vol. 25, n° 28, January 1991, pages 73–77.

[BI93] A.P. Black and M.P. Immel, "Encapsulating Plurality," *European Conference on Object Oriented Programming (ECOOP'93)*, edited by O. Nierstrasz, LNCS, n° 707, Springer-Verlag, July 1993, pages 57–79.

[vdBL91] J. van den Bos and C. Laffra, "Procol: A Concurrent Object-Language with Protocols, Delegation and Persistence," *Acta Informatica*, n° 28, September 1991, pages 511–538.

[BM94] S. Brandt and O.L. Madsen, "Object-Oriented Distributed Programming in BETA," [GNR94], pages 185–212.

[BY 87] J.-P. BRIOT and A. YONEZAWA, "Inheritance and Synchronization in Concurrent OOP," *European Conference on Object Oriented Programming (ECOOP'87)*, edited by J. Bézivin, J.-M. Hullot, P. Cointe and H. Lieberman, LNCS, n° 276, Springer-Verlag, June 1987, pages 32–40.

[BRI 89] J.-P. BRIOT, "Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment," *European Conference on Object Oriented Programming (ECOOP'89)*, edited by S. Cook, Cambridge University Press, July 1989, pages 109–129.

[BRI 96] J.-P. BRIOT, "An Experiment in Classification and Specialization of Synchronization Schemes," to appear in *2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, edited by K. Futatsugi and S. Matsuoka, LNCS, Springer-Verlag, March 1996.

[CIR+93] R. CAMPBELL, N. ISLAM, D. RAILA, and P. MADANY, "Designing and Implementing Choices: An Object-Oriented System in C++," [MEY 93a], pages 117–126.

[CGL+92] R. CAPOBIANCHI, R. GUERRAOUI, A. LANUSSE, and P. ROUX, "Lessons from Implementing Active Objects on a Parallel Machine," *Usenix Symposium on Experiences with Distributed and Multiprocessor Systems*, 1992, pages 13–27.

[CAR 93] D. CAROMEL, "Towards a Method of Object-Oriented Concurrent Programming," [MEY 93a], pages 90–102.

[GGM 94] B. GARBINATO, R. GUERRAOUI, and K.R. MAZOUNI, "Distributed Programming in GARF," [GNR 94], pages 225–239.

[GGM 95] B. GARBINATO, R. GUERRAOUI, and K.R. MAZOUNI, "Implementation of the GARF Replicated Objects Platform," *Distributed Systems Engineering Journal*, February 1995, pages 14–27.

[GFG 96] B. GARBINATO, P. FELBER, and R. GUERRAOUI, "Protocol Classes for Designing Reliable Distributed Environments," To appear in *European Conference on Object Oriented Programming (ECOOP'96)*, edited by P. Cointe, LNCS, Springer-Verlag, Jly 1996.

[GCL+92] R. GUERRAOUI, R. CAPOBIANCHI, A. LANUSSE, and P. ROUX, "Nesting Actions Through Asynchronous Message Passing: the ACS Protocol," *European Conference on Object Oriented Programming (ECOOP'92)*, edited by O. Lehrmann Madsen, LNCS, n° 615, Springer-Verlag, June 1992, pages 170–184.

[GNR 94] R. GUERRAOUI, O. NIERSTRASZ, and M. RIVEILL (editors), *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, LNCS, n° 791, Springer-Verlag, 1994.

[GS 95] R. GUERRAOUI and A. SCHIPER, "The Transaction Model vs Virtual Synchrony Model: Bridging the Gap," *Distributed Systems:*

From Theory to Practice, edited by K. Birman, F. Cristian, F. Mattern and A. Schiper, LNCS, n° 938, Springer-Verlag, 1995.

[GUE 95b] R. GUERRAOUI, "Modular Atomic Objects," *Theory and Practice of Object Systems (TAPOS)*, Vol. 1, n° 2, John Wiley & Sons, November 1995, pages 89–99.

[JÉZ 93] J.-M. JÉZÉQUEL, "Transparent Parallelization Through Reuse: Between a Compiler and a Library Approach," *European Conference on Object Oriented Programming (ECOOP'93)*, edited by O. Nierstrasz, LNCS, n° 707, Springer-Verlag, July 1993, pages 384–405.

[KB 93] M. KARAORMAN and J. BRUNO, "Introducing Concurrency to a Sequential Language," [MEY 93a], pages 103–116.

[KDR+ 91] G. KICZALES, J. DES RIVIÈRES and D. BOBROW, *The Art of The Meta-Object Protocol*, MIT Press, 1991.

[KIC 94] G. KICZALES (editor), "Foil For The Workshop On Open Implementation," "`http://www.parc.xerox.com/PARC/spl/eca/oi/workshop-94/foil/main.html`", October 1994.

[LJP 93] R. LEA, C. JACQUEMOT, and E. PILLEVESSE, "COOL: System Support for Distributed Programming," [MEY 93a], pages 37–47.

[LIE 87] H. LIEBERMAN, "Concurrent Object-Oriented Programming in Act 1," [YT 87], pages 9–36.

[LS 83] B. LISKOV and R. SHEIFLER, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 5, n° 3, 1983.

[LL 94] C.V. LOPES and K.J. LIEBERHERR, "Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications," *European Conference on Object Oriented Programming (ECOOP'94)*, edited by M. Tokoro and R. Pareschi, LNCS, n° 821, Springer-Verlag, July 1994, pages 81–99.

[MAE 87] P. MAES, "Concepts and Experiments in Computational Reflection," *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Special Issue of Sigplan Notices*, Vol. 22, n° 12, December 1987, pages 147–155.

[MAF 95] S. MAFFEIS, "Run-Time Support for Object-Oriented Distributed Programming," *PhD Dissertation*, University of Zurich, Switzerland, February 1995.

[MMA+ 95] H. MASUHARA, S. MATSUOKA, K. ASAI,, and A. YONEZAWA, "Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation," *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), Special Issue of Sigplan Notices*, Vol. 30, n° 10, October 1995, pages 300–315.

[MY 93] S. MATSUOKA and A. YONEZAWA, "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages," *Research Directions in Concurrent Object-Oriented Programming*, edited by G. Agha, P. Wegner and A. Yonezawa, MIT Press, 1993, pages 107–150.

[MGG 95] K. MAZOUNI, B. GARBINATO, and R. GUERRAOUI, "Building Reliable Client-Server Software Using Actively Replicated Objects," *Technology of Object-Oriented Languages and Systems (TOOLS-Europe'95)*, edited by I. Graham, B. Magnusson, B. Meyer and J.-M Nerson, Prentice Hall, March 1995, pages 37–53.

[McA 95] J. MCAFFER, "Meta-Level Programming with CodA," *European Conference on Object Oriented Programming (ECOOP'95)*, edited by W. Olthoff, LNCS, n° 952, Springer-Verlag, August 1995, pages 190–214.

[McH 94] C. MCHALE, "Synchronization in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance," *PhD Thesis*, Dept. of Computer Science, Trinity College, Dublin, Ireland, October 1994. ("`ftp://ftp.dsg.cs.tcd.ie/pub/doc/dsg-86b.ps.gz`").

[MEY 93a] B. MEYER (editor), "Concurrent Object-Oriented Programming," Special Issue, *Communications of the ACM (CACM)*, Vol. 36, n° 9, September 1993.

[MEY 93b] B. MEYER, "Systematic Concurrent Object-Oriented Programming," [MEY 93a], pages 56–80.

[NWM 93] J. NICOL, T. WILKES, and F. MANOLA, "Object-Orientation in Heterogeneous Distributed Computing Systems," *IEEE Computer*, Vol. 26, n° 6, June 1993, pages 57–67.

[OI 94] H. OKAMURA and Y. ISHIKAWA, "Object Location Control Using Meta-Level Programming," *European Conference on Object Oriented Programming*, edited by M. Tokoro and R. Pareschi, LNCS, n° 821, Springer-Verlag, July 1994, pages 299–319.

[PS 88] G.D. PARRINGTON and S.K. SHRIVASTAVA, "Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems," *European Conference on Object Oriented Programming (ECOOP'88)*, edited by S. Gjessing and K. Nygaard, LNCS, n° 322, Springer-Verlag, 1988, pages 234–249.

[ROZ 92] M. ROZIER, "Chorus," *Usenix International Conference on Micro-Kernels and Other Kernel Architectures*, 1992, pages 27–28.

[THO 92] L. THOMAS, "Extensibility and Reuse of Object-Oriented Synchronization Components," *International Conference on Parallel Languages and Environments (PARLE'92)*, LNCS, n° 605, Springer-Verlag, June 1992, pages 261–275.

[WEG 87] P. WEGNER, "Dimensions of Object-Based Language Design," *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Special Issue of Sigplan Notices*, Vol. 22, n° 12, December 1987, pages 168–182.

[WEI 89] W. WEIHL, "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types," *ACM Transactions on Programming Languages and Systems*, Vol. 11, n° 2, 1989.

[WIN 94] J. WING, "Decomposing and Recomposing Transaction Concepts," [GNR 94], pages 111–122.

[YOK 92] Y. YOKOTE, "The Apertos Reflective Operating System: The Concept and its Implementation," *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92), Special Issue of Sigplan Notices*, Vol. 27, n° 10, October 1992, pages 414–434.

[YT 87] A. YONEZAWA and M. TOKORO (editors), *Object-Oriented Concurrent Programming*, Computer Systems Series, MIT Press, 1987.