Concurrency and Distribution in Object-Oriented Programming *

Jean-Pierre BRIOT
Laboratoire d'Informatique de Paris 6,
UPMC - Case 169, 4 place Jussieu,
75252 Paris Cedex 05, France.

Rachid GUERRAOUI
Département d'Informatique, École Polytechnique Fédérale de Lausanne
CH-1015, Lausanne, Switzerland.

Klaus-Peter LÖHR
Institut für Informatik, Freie Universität Berlin
D-14195 Berlin, Germany.

Abstract

This paper aims at classifying and discussing the various ways along which the object paradigm is used in concurrent and distributed contexts. We distinguish the *library* approach, the *integrative* approach, and the *reflective* approach. The library approach applies object-oriented concepts, as they are, to structure concurrent and distributed systems through class libraries. The integrative approach consists in merging concepts such as: object and activity, message passing and transaction, etc. The reflective approach integrates class 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.

Technical Report - May 1998

Laboratoire de Systèmes d'Exploitation
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Switzerland

*To appear in ACM Computing Surveys, September 1998.

1 Introduction

It is now well accepted that the object paradigm provides good foundations for the new challenges of concurrent and distributed computing. Object notions, rooted in the *data abstraction* principle and the *message passing* metaphor, are strong enough to structure and encapsulate modules of computation, and flexible enough to match various granularities of software and hardware architectures.

Most of object-based programming languages do have some concurrent or distributed extension, 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)* standardisation initiatives for heterogeneous distributed computing, are based on object concepts [Nicol et al. 1993, OMG 1995, Mowbray and Zahavi 1995].

As a result, many object-based concurrent 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, customised, integrated, expanded) with concurrency and distribution challenges and current technology. Rather than an exhaustive study of various object-oriented concurrent and distributed programming systems, this paper aims at classifying and discussing the various ways along which the object paradigm is used in concurrent and distributed contexts.

1.1 A Coarse Classification in Three Approaches

By analysing current experiences and trends (as for instance reported in [Meyer 1993]), we have distinguished three main approaches in object-based concurrent and distributed programming: the *library* approach, the *integrative* approach and the *reflective* approach. This paper discusses and illustrates these three approaches in turn.

The *library* approach (Sect. 2) applies object-oriented concepts, as they are, to structure concurrent and distributed systems through class libraries. Various components, such as processes, synchronisation constructs, files, and name servers, are represented by various object classes. This approach provides genericity of the software architectures. Programming remains mostly standard (sequential) object-oriented programming. Roughly speaking, the basic idea is to extend the library rather than the language.

The *integrative* approach (Sect. 3) consists in unifying concurrent and distributed system concepts with object-oriented ones. For example, merging the notions of *process* and *object* gives rise to the notion of *active object*, and merging the notions of *transaction* and *object invocation* gives rise to the notion of *atomic invocation*. However, integration is not always that smooth. We will see that concepts may be in conflict, notably inheritance with synchronisation and replication with communication (Sect. 3.6).

The *reflective* approach (Sect. 4) integrates protocol libraries within an object-based programming language. The idea is to separate the application program from the various aspects of its implementation and computation contexts (models of computation, communication, distribution etc.), which are described in terms of *meta-programs*. This enables (dynamic) system customisation with minimal impact on the application programs.

1.2 Complementarity of the Approaches

Although these approaches may appear to be in conflict at first glance, in fact they are not. More precisely, the research activities taking these approaches have complementary goals.

The *library* approach is oriented towards system builders and aims at identifying basic concurrent and distributed abstractions. This approach provides services and constructs for building an object-based concurrent or distributed model.

The *integrative* approach is oriented towards application builders and aims at defining a high level programming language with few unified concepts. This approach assumes an object-based concurrent or distributed model that describes how the services interact.

The *reflective* approach is oriented towards both application builders and system builders. It may be considered as a "bridge" between the two previous approaches as it helps to transparently integrate various computing protocol libraries within a programming language/system. Moreover, it helps at combining the two other approaches by making explicit the separation between their respective levels (i.e., the integrative approach for the end user, and the library approach for developing and customising the system). The success of a reflective system relies both on a high level programming language and on a rich library of concurrent and distributed programming abstractions.

It is important to notice that the three approaches do not correspond to disjoint categories of languages and systems. As we will point out in the paper (e.g., in Sect. 5.4), several languages and systems were built following more than one approach.

1.3 Concurrency and Distribution

Before presenting our classification in more detail, we first briefly clarify our distinction between the terms *concurrency* and *distribution*.

There are different ways of running a concurrent program on an execution platform. The program may be executed on a uniprocessor, e.g., using a threading system, or on a parallel computer. Thus, while concurrency is a *semantic* property of a program, parallelism pertains to its *implementation*, as determined by the compiler and other systems software.

In contrast to parallelism which may usually be seen as the implementation of concurrency, distribution is rather an independent notion. First, distribution does not necessarily imply concurrency: a purely sequential program may be executed across machine boundaries, using remote procedure calls, i.e., in a distributed fashion. A similar situation is found with client/server systems: while a server may or may not be concurrent, its clients rarely are; only when we view a server and its clients as one system, we see a concurrent system operating in a distributed fashion. Second, distribution intrinsically implies independent failures, i.e., part of a program might stop running (because of a crash failure), whereas the rest of the program might still be running. In a concurrent but not distributed context, it is usually assumed that programs have total failure semantics: either the complete program is running, or none of the parts is.

1.4 Previous Work

We assume that the reader is familiar with traditional concurrency and distribution concepts. It should also be kept in mind that object-based concurrent and distributed programming are well-established disciplines which are supported by many languages. One the one hand, *Ada* [Ada 1983] is a well-known example of a concurrent object-based language, and *SR* [Andrews et al. 1993], although less known, features versatile and powerful concurrency constructs. On the other hand, *Argus* [Liskov and Sheifler 1983] and *Emerald* [Black et al. 1987] are widely known for their distribution support.

Combining concurrency and distribution with object-orientation proper, i.e., including inheritance, has been the subject of many research projects since 1985. Several new language designs, representing the integrative approach, are discussed and compared in [Papathomas 1989] and [Papathomas 1995]. An early book featuring different articles on concurrent object-oriented programming is [Yonezawa and Tokoro 1987]. A more recent one is [Agha et al. 1993]. Furthermore, several workshops have been devoted to object-based concurrency and distribution: the proceedings of these workshops can be found in [Agha et al. 1989, Agha et al. 1991, Tokoro et al. 1992, Guerraoui et al. 1993, Briot et al. 1995].

2 The Library Approach

2.1 Modularity and Structuring Needs

The basic idea of the *library* approach is to *apply* encapsulation and abstraction, and possibly also class and inheritance mechanisms, as a structuring tool to design and build concurrent and distributed computing systems. In other words, the issue is to build and program a concurrent or distributed system, with a given object-oriented methodology and a given object-oriented programming language. The main motivation is to increase modularity, by decomposing systems in various components with clear interfaces. This improves structuring of concurrent and distributed systems, as opposed to uniform single layer systems, in which the different levels of abstraction are difficult to distinguish and understand.

Applied to distributed operating systems, the *library* approach has led to a new generation of systems, such as *Chorus* [Rozier 1992] and *Choices* [Campbell et al. 1993], based on the concept of *micro-kernel*. The architecture of the (generic) distributed operating system is organised along abstract notions of class components, which may then be specialised for a given instantiation/porting of the (virtual) system. 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 library approach, we survey examples of libraries for concurrent as well as for distributed programming in two different well-known object-oriented languages: (1) the *Smalltalk-80*

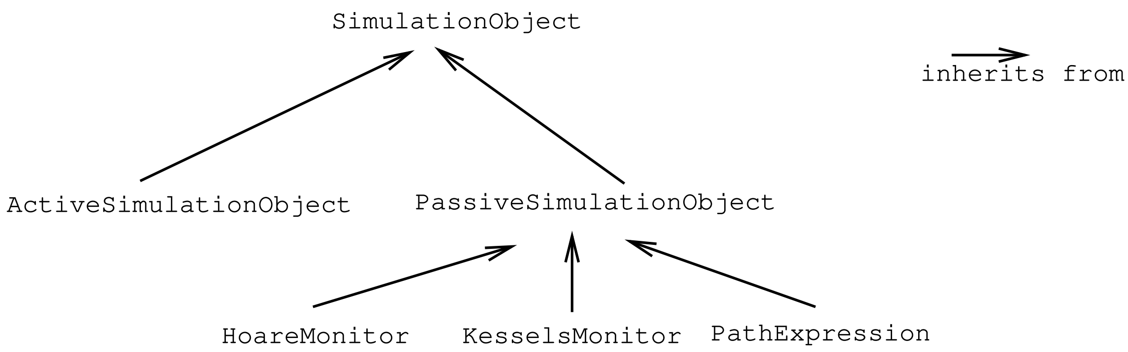

Figure 1: Hierarchy of concurrency classes in Simtalk.

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; and (2) *C++*, whose widespread use has resulted in a proliferation of concurrency libraries [Wilson and Lu 1996].

2.2 Smalltalk Libraries

Smalltalk is often considered as one of the purest examples of object-oriented languages [Goldberg and Robson 1989]. 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*, which describe and implement various programming constructs (e.g., control structures and data structures), internal resources (e.g., messages, processes and compiler), and a sophisticated programming environment with integrated tools (e.g., browser, inspector and debugger).

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 `BlockClosure`, are essential for building various control structures that the user may extend as required.

2.2.1 Libraries for concurrent programming

In Smalltalk, blocks are also the basis for multi-threaded concurrency through *processes*. The standard class `Process` describes their representation, and its associated methods implement their management (suspend, resume, adjust priority). The behavior of the process scheduler is itself described by a class named `ProcessorScheduler`. The basic synchronisation 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.

Thanks to this uniform approach, concurrency concepts and mechanisms are well encapsulated and organised 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 construct more sophisticated abstractions on top of the basic standard library of concurrency classes, as for example in the *Simtalk* [Bezivin 1987] or *Actalk* [Briot 1989] platforms. The Simtalk platform implements and classifies various synchronisation and simulation abstractions (pessimistic or optimistic simulation objects, Hoare monitors, Kessels monitors, etc.) on top of Smalltalk standard abstractions/classes. A sample of the hierarchy of Simtalk classes is shown in Figure 1. Within the Actalk project for instance, the standard scheduler was extended to parametrise and classify various scheduling policies.

2.2.2 Libraries for distributed programming

Smalltalk 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

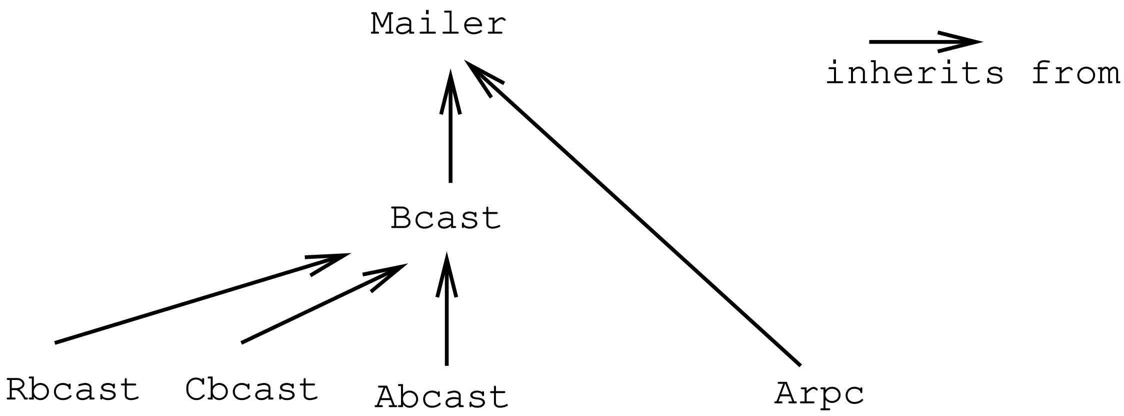

Figure 2: Communication in GARF.

standard Smalltalk *Binary Object Streaming Service (BOSS)* library provides a basic support for building distribution mechanisms, e.g., marshaling and transactions. The *HP Distributed Smalltalk* product provides a set of distributed services following the OMG (*CORBA*) standard [OMG 1995], themselves implemented as Smalltalk-80 class libraries.

Projects like *GARF* [Garbinato et al. 1994, Garbinato et al. 1995] and *BAST* [Garbinato et al. 1996, Garbinato and Guerraoui 1997] go a step further in providing abstractions for reliable distributed programming. In GARF, two complementary class hierarchies have been developed for various communication models (e.g., point-to-point, multicast and total order multicast) and object models (e.g., monitor, persistent, and replicated). For instance, class `Mailer` implements remote message passing. Class `Abcast` (a subclass of `Mailer`) broadcasts an invocation to a set of replicated objects, and ensures that the messages are totally ordered (ensure the consistency of the replicas). These classes constitute the adequate support for the development of fault-tolerant applications where critical components are replicated on several nodes of a network. A sample of the hierarchy of GARF classes is shown in Figure 2.

The BAST project aims at building abstractions at a lower level. Roughly speaking, BAST provides distributed protocols, such as total order multicast and atomic commitment, that are used in the implementation of GARF classes. For instance, BAST supports classes `UMPObject` for unreliable message passing, and subclasses `RMPObject`, and `RMCObject`, respectively, for reliable message passing and reliable multicast communication [Garbinato et al. 1996, Garbinato and Guerraoui 1997].

2.3 C++ Libraries

As opposed to Smalltalk, C++ is not genuinely object-oriented [Stroustrup 1993]. It is an object-oriented extension of C, a language originally designed for systems programming. Thus, C++ is not the ideal vehicle for building object-oriented applications. Nevertheless, it has become the most widely used object-oriented language, and it is *the* language for object-oriented systems programming. As a consequence, building concurrency and distribution libraries in C++ has been more than a marriage of convenience. As explained in Section 1.2, the systems programmer needs flexibility and therefore prefers libraries to built-in features. That programmer also likes to exploit the low-level mechanisms and services offered by the underlying execution platform. As the library approach allows for any functionality of a given platform to be wrapped in C++ functions or classes, it is not surprising that there is a wide variety of concurrency and distribution mechanisms cast in C++ libraries. In fact, any programmer can readily build wrappers for concurrency and distribution mechanisms from her favorite platform.

2.3.1 Libraries for concurrent programming

Class libraries can be built for all kinds of process concepts, heavy-weight or light-weight, and for their corresponding synchronisation mechanisms. Many concurrent programs are conveniently implemented using a threading system (e.g., network servers, interactive programs, parallel programs). In the following, we first look into object-oriented threading libraries.

```
class producer: public task {
public:
      producer()
      { .....          // compute x
        results(x);
      }
}

int main()
{ producer p;
  .....         // compute y
  cout << "Results are " << p.result() << " and " << y;
  return 0;
}
```

Figure 3: Customized Sun C++ `task` object.

Representation of threads. While defining classes for synchronisation objects such as semaphores is a straightforward exercise, it is not obvious how to cast a thread abstraction into a class. There are at least three different ways, depending on how the activity of a thread object is described:

1. A thread is an instance of a subclass of some class `Thread`, and the activity of the thread is described by the *constructor* of the subclass. This is akin to the Simula approach to coroutines [Birtwistle et al. 1973]: the body of a coroutine class describes both initialisation and activity of a coroutine object.
2. A thread is an instance of a subclass of class `Thread`, but its activity is described by overriding a *special method.*
3. A thread is an instance of class `Thread`, and its activity is described by a function that is passed as a *parameter* to the constructor or a special method.

In all these approaches, creating a thread object spawns a new thread. Note that the lifetime of its activity may be shorter than the lifetime of the object itself.

An example of the first approach is the *coroutine* part of *Sun's C++ library* [Sun 1995]. The library offers a class `task` which plays the role of the class `Thread` in the classification above. A `task` object is implemented as a coroutine, with non-preemptive scheduling. There is also a class `Interrupt_handler` that allows to catch Unix software interrupts (signals). Typical operations on tasks are `result()` (wait for termination), `rdstate()` (get state), etc. Synchronisation is supported by low-level wait operations and by object queues.

Figure 3 shows a fragment of a simple program using the coroutine library. The main program, by declaring the object `p`, creates a task which executes the `producer()` constructor. There is no interaction between parent and child task, except that the child terminates producing a result, which is picked up by the parent.

An example of the third approach is found in *PRESTO*, a system for parallel programming on a multiprocessor [Bershad et al. 1988]. A newly created thread is idle until explicitly started. The function to be executed and its parameters are passed as parameters to the `start` operation. For synchronisation, PRESTO features atomic integers and lock, monitor and condition classes.

An alternative can be found in *DC++*, where the function to be executed is passed to the constructor. DC++ [Schill and Mock 1993] is a system for distributed execution of C++ programs on top of *DCE*, the OSF Distributed Computing Environment [OSF 1994]. While DC++ focuses on distribution, it does also offer a few classes for concurrent programming. Concurrency is implemented by using the DCE threading subsystem. Thus, DC++ is readily ported to any system that is equipped with the DCE platform. The DC++ library includes a class `Thread` as described above, plus a few classes for synchronisation. Parameters of the `Thread` constructor allow the user to choose among different scheduling policies.

The ACE library. ACE stands for Adaptive Communications Environment [Schmid 1995]; it is a toolkit for developing communication-oriented software. One of the goals of the ACE threading library is to present abstractions that subsume the threading mechanisms of different platforms (POSIX, Solaris 2, Win32), thus enhancing portability.

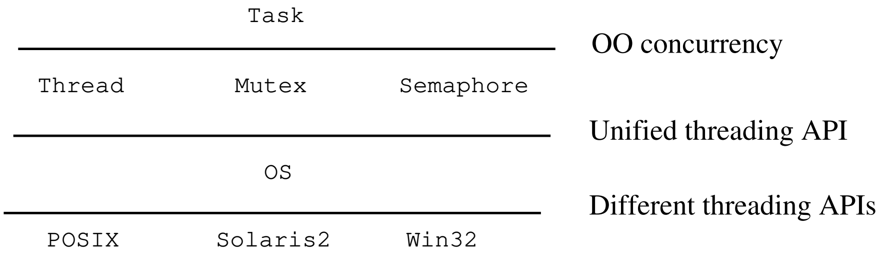

Figure 4: Architecture of the ACE concurrency library.

```
typedef void *(*THR_FUNC)(void *);

class Thread {
public:
static int spawn(THR_FUNC fun,             // create thread to execute fun
                 void *arg,                // with argument arg
                 long flags,
                 thread_t * = 0,
                 void *stack = 0,
                 size_t stack_size = 0,
                 hthread_t *t_handle = 0); // to be referred to by t_handle

static int suspend(hthread_t);             // suspend thread

static void exit(void *status);            // terminate current thread

.......                                    // more routines
}
```

Figure 5: Class `Thread` in ACE.

Figure 4 shows part of the layered architecture of the ACE concurrency class library. Portability of the concurrency classes is achieved through a class `OS` that just packages threading-related functions, hiding the peculiarities of different native threading systems.

ACE has classes `Mutex, Semaphore, RW_Mutex` and others for synchronisation. A class template `Guard` is parametrised with a lock class (e.g., `Mutex`). A guard object acquires and releases a lock upon initialisation and finalisation, respectively, similar to a PRESTO monitor object; thus, declaring a guard in a block will turn this block into a critical region. (Note that ACE guards have nothing to do with the Boolean expression guards used in genuinely concurrent languages.)

Threads are handled on a very low level of abstraction in ACE. There does exist a class `Thread` (Fig. 5), but this is just a package of static functions such as `spawn, join, yield` etc., abstracting from the idiosyncrasies of the threading functions of POSIX, Solaris and Win32. Another class, `Thread_Manager`, does serve the purpose of creating and using thread manager objects; they are responsible for managing groups of threads, spawning new members, disposing of a thread when it terminates etc. But there is no class that would resemble Smalltalk's `Process` or the `task` from Figure 3.

A relatively high-level concept in ACE is the `Task` class. This class must not be confused with Sun's `task` class mentioned above. `Task` is an abstract class whose interface is designed for use according to the Stream/Module concept for layered communication. Subclass objects of `Task` can participate in a batch of modules implementing a stream. Each task must provide a `put` operation to be invoked from an adjacent module in a stream and an `svc` operation ("service") for asynchronous execution of the invoked service in case of an active task object.

The AVTL library. The challenge of object-oriented programming for parallel computing systems is to find an object model that fits in with the preferred models for parallel computation. For a library-based solution there is no choice - the object model is given by the sequential language. Here the most straightforward path to parallel processing is just executing concurrent programs with threads on a shared-memory multiprocessor, as mentioned above for PRESTO. This produces functional parallelism, but no data parallelism.

The Amelia Vector Template Library (*AVTL*) [Sheffler 1996] is an example of library support for parallel processing of vectors. While an approach like AVTL is tailored towards a specific class of applications, it has the advantage of hiding communication from the programmer. If we are willing to pay the price of low-level message-based programming, unlimited flexibility is achieved by libraries that connect to a communication platform, e.g., MPI [Skjellum et al. 1996]. Libraries of this kind can be seen as the "parallel" equivalent to threading libraries as described above.

2.3.2 Libraries for distributed programming

We have seen that for C++, the library approach tends to mirror the functionality of the underlying execution platform. This is not only true for concurrency but also for distribution. So we often find library classes that encapsulate remote communication mechanisms such as ports or sockets (e.g., ACE supports Unix socket objects).

DC++. The DC++ system mentioned above supports remote object invocation. Note, however, that distribution and concurrency are not strictly orthogonal in DC++. Remote invocation comes in two flavours: synchronous and asynchronous (where asynchrony leads to truly parallel execution of client and server). Asynchronous invocation of local objects is not directly supported. Ironically, this implies that it is easier in DC++ to write a concurrent or distributed program than to write a centralised one.

Choices. Choices [Campbell et al. 1993] is a *generic* operating system. Its 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 a generalised notion of pointer. Class `ObjectStar` provides transparency for remote communications without need for a pre-compilation step. This class is also used by 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 project shows that a distributed operating system, developed with an object-oriented methodology and programming language (C++ in this case), helps at achieving better genericity and extensibility.

Peace. Similar in spirit to Choices, the *Peace* parallel operating system [Schröder-Preikschat 1994] is targeted at distributed memory multicomputers. Like Choices, Peace is actually a *family* of object-oriented operating systems. Its components, implemented in C++, can be configured in different ways, in order to fit different hardware platforms and offer varying functionality.

Peace makes heavy use of inheritance in implementing the system family concept. A stepwise bottom-up design of minimal extensions using subclasses results in a fine-grain inheritance hierarchy. Exploiting this scheme, application programs interface to the operating system by simply extending certain system classes.

The basic unit of concurrent execution, the *thread*, is introduced through a series of abstractions. Most threads are made up from two objects, of classes `native` and `thread`, respectively. The class `native` describes the kernel-level part of the thread, and class `thread` refers to the user-level part. An application program can declare a subclass of `thread`, say, `custom`, redefining the method `action()` inherited from class `thread`. Creating a `custom` object causes the creation of a thread that executes the redefined `action()`.

2.4 In Search for Standard Abstractions

The main issue underlying the library approach is the design and implementation of adequate abstractions on top of which various higher-level concurrency and distribution abstractions can be built in a convenient way.

One of the most significant examples 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 synchronisation for concurrent programming. Such a basic abstraction may be used as a foundation to build various higher-level synchronisation mechanisms (e.g., the `Guard` class of ACE). Classification and specialisation mechanisms, as offered by object-oriented programming, are then appropriate to organise such a library/hierarchy of abstractions, as for instance in the Simtalk platform (Sect. 2.2). Peace is a typical example for an extremely careful design of a hierarchy of thread abstractions.

An example of developing concurrency abstractions complementary to program abstractions can be found in the *Demeter* environment [Lopes and Lieberherr 1994]. 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 recently been added: the "concurrency block", which describes the abstract synchronisation patterns between processes. The abstract specifications and the relative independence between these three components are intended to help with the development of generic and reusable programs.

An example of developing standard libraries for the basic support of distributed programming may be found in the *Beta* programming language [Brandt and Lehrmann Madsen 1994, Lehrmann Madsen et al. 1993]. 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. This approach enables the programmer to add distributed features to a given sequential/centralised program without changing the program logic, i.e., through additions rather than changes [Brandt and Lehrmann Madsen 1994, page 199].

In [Black 1991] the transaction concept is decomposed 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 concurrency control constraints as strong as those required for a banking application[1].

A fundamental study of abstractions for reliable distributed programming can be found in [Schneider 1985]. A similar approach was adopted in [Guerraoui and Schiper 1996] where *consensus* is suggested as a basic building block for unifying many agreement problems as well as their underlying paradigms (e.g., transactions and view synchronous communication [Guerraoui and Schiper 1995]). A complete implementation framework is described in [Garbinato and Guerraoui 1997].

3 The Integrative Approach

3.1 Unification Needs

The amount of issues and concepts required represents one of the major difficulties of concurrent and distributed programming. In addition to classical constructs of sequential programming, concurrent and distributed computation introduces concepts such as *process*, *semaphore*, *monitor* and *transaction.* The library approach helps at structuring concurrency and distribution concepts and mechanisms, but it keeps them disjoint from the objects structuring the application programs. In other words, the programmer still faces at least two major different issues: programming with objects *and* managing concurrency 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 concurrency and distribution aspects (and more specifically the manipulation of the objects implementing them) add up to the standard programming style. For instance, a library implementing asynchronous and remote communication in Eiffel will force the programmer with some amount of explicit message manipulation (see [Karaorman and Bruno 1993]), as opposed to standard implicit message passing. One may then choose to *integrate* such constructs directly into the programming language as the extension of a standard language, e.g., *Eiffel//* [Caromel 1990], or to define a brand new language containing that construct.

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

[1] The latter application requires strict serialisation of transactions (e.g., through a locking mechanism) while the former does not.

3.2 Dimensions of Integration

There are various possible dimensions of integration between object-oriented programming concepts and concurrency and distribution concepts. We distinguish here three main different dimensions. Note that they are relatively independent and, as we will see, a given language or system may follow one dimension of integration but not another one.

1. A first dimension of 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[2]. Actor languages [Lieberman 1987, Agha 1986] are typical examples of programming languages based on the notion of an active object. Objects that are not active are sometimes called *passive*.
2. A second dimension of integration associates *synchronisation* with *object activation*, leading to the notion of a *synchronised object*. Message passing is then considered as an implicit synchronisation 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 synchronised object, as the existence of a (single) activity private to the object actually enforces the serialisation of invocations. However, some languages or systems, e.g., *Guide* [Balter et al. 1994] or *Arjuna* [Parrington and Shrivastava 1988], associate synchronisation to objects although they distinguish the notions of object and autonomous activity. Another example is *Java* [Lea 1997], where a new private lock is implicitly associated with each newly created object.
3. A third dimension 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. The message passing metaphor is seen as a transparent way of invoking either local or remote objects. The *Emerald* distributed programming language [Black et al. 1987] 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, in order to support inter-object synchronisation and fault-tolerance [Liskov and Sheifler 1983, Guerraoui et al. 1992].

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, was quite influent [Yonezawa and Tokoro 1987], following the way traced by actor-languages [Lieberman 1987, Agha 1986].

3.3.1 Levels of object concurrency

The independence of object activities provides what is usually called *inter-object* concurrency. Some languages such as *POOL* [America and van der Linden 1990] provide only this level of concurrency. In several computation models however (e.g., *Actors* [Agha 1986]), an active object is allowed to process several requests simultaneously, thus owning more than one internal activity: this is called *intra-object* concurrency.

More generally, one may consider different levels of object concurrency. Our classification below is a slight refinement extension of Peter Wegner's classification [Wegner 1990]. An active object may be:

- *serial*[3]: only one message is computed at a time. Example of languages offering serial active objects are *POOL* [America 1987] and *Eiffel//* [Caromel 1990].
- *quasi-concurrent*: several method activations may coexist, but at most one of them is not suspended. (This is similar to a monitor using event variables to suspend processes.) Examples of languages offering quasi-concurrent objects are: *ABCL/1* [Yonezawa et al. 1986] and *ConcurrentSmalltalk* [Yokote and Tokoro 1987].
- *concurrent*: There is true intra-object concurrency, but some degree of control, i.e., restrictions, apply, as specified by the programmer. Examples of languages offering concurrent objects are: actor languages, e.g., *ACT++* [Kafura and Lee 1990] and also *CEiffel* [Lohr 1993].

[2]This similarity has been noted, e.g., in [Meyer 1993].
[3]This is also often called *atomic* object.

- *fully concurrent*: concurrency within the object is not restricted. This usually means that such objects are functional (they have no state or at least no changing state). Actor languages support such fully concurrent objects and they are called *unserialised actors*.

Setting the possible level of object internal concurrency for a programming language is a matter of design decision, and an important decision. Some researchers have argued that intra-object concurrency should be banned altogether because reasoning about programs that contain serial objects only is much easier [Meyer 1993]. But single-threaded active objects are prone to the same pitfalls as nested monitors, although the problem is mitigated when using asynchronous invocation. Allowing intra-object concurrency increases the expressive power as well as the overall concurrency, but requires some additional concurrency control in order to ensure object state consistency, and some careful management of resources in order to maintain efficient implementations.

Finally, note that for efficiency reasons, fully concurrent objects are usually implemented as passive objects (i.e., standard objects without any activity) without any synchronisation, and replicated onto every processor. Thus every invocation from an active object is immediately processed in the resource (process) of the sender.

3.3.2 Reactivity vs autonomy

One of the important characteristics of object-oriented programming is the *reactivity principle* (as stated by Alan Kay in the late 60's [Kay 1969]). An object is said to be *reactive* in the sense that it reacts to an event (when receiving a message). Moreover the only way to activate an object is by sending a message to it. This is opposed to the idea of a process which starts processing as soon as created.

The integration of object with process (the concept of active object) raises the issue whether reactivity will be preserved, or shadowed by the autonomous behavior of the process. We can distinguish two kinds of active objects:

- *reactive active object*: it keeps up with the reactivity principle, and thus can be activated only through message passing, e.g., as in ACT++.
- *autonomous active object*: it may compute before being sent a message, e.g., as in POOL and Eiffel//.

Although the models are opposite, they may very easily simulate each other. A reactive active object having a method whose body is an endless loop will turn into an autonomous active object after receiving a corresponding message. An autonomous active object whose activity is to keep accepting incoming messages actually models a reactive active object. (See the example of POOL in Sect. 3.3.4 where this is actually the default case.)

3.3.3 Implicit vs explicit acceptance of messages

Another issue which is related to the reactivity of active objects is whether there should be implicit or explicit acceptance of messages (or even both). Implicit acceptance means that a message is automatically accepted eventually[4] after it has been received. Explicit acceptance means that the object explicitly states that it is willing to accept a certain pattern of message. This is analogous to the accept statement in *Ada*.

By looking at the relation between the reactivity vs autonomy issue and the implicit vs explicit acceptance issue, the first impression could be that reactivity always implies implicit acceptance, and that autonomy always implies explicit acceptance. This is often the case, but not always, as described below:

- ABCL/1 is an example of a programming language based on the concept of reactive active object and which offers explicit message acceptance from within a method.
- CEiffel supports autonomous objects. But methods are activated implicitly, independently of any explicit message acceptance (see Sect. 3.3.5).
- a synchronisation scheme, named *abstract states*, which we will describe later (in Sect. 3.4.3), combines implicit acceptance with some explicit matching of a message.

[4] The actual processing may be delayed after the message is received because of synchronisation requirements.

```
CLASS Queue
      .....
METHOD enq(item: T)
BEGIN  cell!put(rear,item);
       rear := (rear+1)MOD size END enq

METHOD deq(): T
BEGIN  RESULT cell!get(front);
       front := (front+1)MOD size END deq

BODY DO IF    empty  THEN ANSWER(enq)
        ELSIF full() THEN ANSWER(deq)
        ELSE              ANSWER ANY FI OD YDOB
END Queue
```

Figure 6: Active queue in POOL.

3.3.4 The concept of body

Most languages following the autonomous active object model are based on the concept of a *body*. There is some distinguished centralised operation (the *body*), which explicitly describes the types and the sequence of requests that the object might accept during its activity. This concept is actually a direct offspring of *Simula-67* [Birtwistle et al. 1973] concept of *body*, which included support for coroutines. This initial potential of objects for concurrency was abandoned, however, both for technological and cultural reasons, by most of the Simula-67 followers.

The concept of body and explicit acceptance is close to the Ada tasking model. The body of an Ada task encapsulates state variables and a statement sequence that begins executing as soon as the task is created. A set of *entries* - comparable to operation signatures - is associated with a task. A remote invocation - looking like a procedure call - refers to one of these entries. A task uses *explicit accept* statements for accepting invocations and executing the requested service.

POOL[5] and Eiffel// [Caromel 1993] are typical representatives of the body concept. A POOL class for active `Queue` objects is shown in Figure 6. The declarative part for local data is omitted. Operations such as `METHOD enq` are declared just like for passive objects. A `Queue` object has a single thread of control. Its activity is described by the statements enclosed in the `BODY/YDOB` keywords (`DO/OD` is an infinite loop). As opposed to Ada, the accept statements, starting with the keyword `ANSWER`, just refer to one or more operation names (`ANY` meaning all operation names).

A queue is in fact usually implemented as a passive object. It is only for demonstration purposes that we present an "active queue". And it should be kept in mind that the body of a POOL class can of course be of arbitrary complexity. Notice that a missing body defaults to `DO ANSWER ANY OD` which actually models a reactive active object.

In a very similar spirit, Eiffel// has a predefined class `PROCESS`. Instances of a (direct or indirect) subclass of `PROCESS` are active objects. The object body is represented by a routine `Live` which has a default implementation in `PROCESS` and is usually redefined in subclasses of `PROCESS` (comparable to POOL's `BODY`). Several other routines inherited from `PROCESS` enable an active object to control the acceptance of invocations in its `Live` routine, much like it is done with `ANSWER` in the POOL language.

3.3.5 Autonomy without body

Although most autonomous active objects models are based on the concept of body with explicit acceptance statements, alternatives are possible. In CEiffel, operations can be specified as autonomous using the *autonomy annotation* `-->--`.[6]

An autonomous operation is executed repeatedly, without being invoked. More precisely, when an autonomous operation finishes, it is implicitly invoked anew. The scheduling mechanism does not dis-

[5] Actually, there are three different versions of Pool: POOL-T, POOL2 and POOL-I

[6] Note that a CEiffel annotation always starts with the characters `--`, just like an Eiffel comment, but is identified as an annotation by the next character. Being comments, annotations are ignored by a standard Eiffel compiler, but are recognized by a specific CEiffel precompiler [Lohr 1993].

```
CLASS Moving CREATION init
FEATURE -- interface
        position: Vector;

        setVelocity(v: Vector) IS
        DO velocity.set(v.x,v.y) END;

FEATURE {} -- hidden
        velocity: Vector;
        stepTime: Real;

        step IS -->--
        DO position.set(position.x + velocity.x*stepTime,
                        position.y + velocity.y*stepTime) END;

        init(startingPoint: Vector; timeUnit: Real) IS
        DO position := startingPoint;
           stepTime := timeUnit  END
END -- Moving
```

Figure 7: Modelling autonomous moving objects in CEiffel.

tinguish between explicit and implicit invocations. Note that the degree of intra-object concurrency will be controlled in CEiffel by other kind of annotations: compatibility annotations (see later in Sect. 3.4.3).

Figure 7 shows the example of class `Moving` which models objects moving autonomously in the plane.

3.4 Synchronised Objects

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

3.4.1 Synchronisation at the message passing level

A straightforward mapping of the message passing mechanism from a sequential computing context to a concurrent one leads to the implicit synchronisation of the sender to the receiver. 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 as it has sent 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. Note that this implies associating an active object with a mail queue that buffers incoming messages (usually in the order of their arrival) before the active object is ready to process them.

Finally, there are languages (e.g., ABCL/1 and ACT++) that introduce some mixed kind of transmission, which immediately returns a promise for (i.e., a handle to) a future reply, without waiting for the actual completion of the invocation. It is thus possible to decouple invocation and waiting for a result. Only when the sender (client) really needs the result - i.e., is going to operate on it - synchronisation with the service provider is required. Integration of futures into the invocation mechanism has the effect that the strict synchronisation inherent in synchronous invocation is replaced with *synchronisation by need*, sometimes called *lazy synchronisation*. The concept of future originated in the actor languages and was included as a specific type of message transmission in early object-oriented concurrent languages, notably ABCL/1[7]. It was later integrated into some extensions of existing languages, like Eiffel// [Caromel 1990]

[7]ABCL/1 actually offers three types of message transmission: synchronous (called *now* type), asynchronous (called *past*

```
server: ActiveServer;
result: R;
.....
result := server.service(args);
.....          -- client continues immediately
result.op      -- synchronisation is implicit
```

Figure 8: Synchronisation by need in Eiffel//.

where it is known as *wait-by-necessity*.

Service execution is *always* asynchronous in Eiffel//. If there is a result, lazy synchronisation takes effect. In Figure 8, the client may proceed immediately after the invocation, becoming blocked only when it tries to prematurely invoke the result object.

3.4.2 Synchronisation at the object level

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

In case of serialised 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 distinguish below three different levels of synchronisation 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 synchronisation. In case of *intra-object* concurrency (i.e., an object processing several requests simultaneously), 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[8]. The typical example is the readers and writers problem, where several readers are free to access simultaneously a shared data. However, the presence of one writer excludes all others (writers and readers). Note that intra-object synchronisation is the equivalent at the object level of what is called (mutual) exclusion synchronisation at the level of data.

Behavioral synchronisation. 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 longer full. This makes synchronisation of services between objects fully transparent. Behavioral synchronisation is the equivalent at the object level of what is called condition synchronisation at the level of data.

Inter-objects synchronisation. 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 synchronisation is not sufficient, and a notion such as atomic transaction [Bernstein et al. 1987], is needed to coordinate the different invocations.

3.4.3 Synchronisation schemes

Various *synchronisation 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 or less integrated within an object-oriented concurrent programming framework.

We can make a general distinction whether synchronisation specifications are centralised or not:

- *centralised* schemes, such as *path expressions*, specify in an abstract and centralised way the synchronisation of the object. As they are centralised, they tend to be naturally associated and then integrated with the class.

type), and with eager-reply (called *future* type)

[8] Note that the case of a mutual exclusion between all methods subsumes the case of a serialised object (as defined in Sect. 3.3.1).

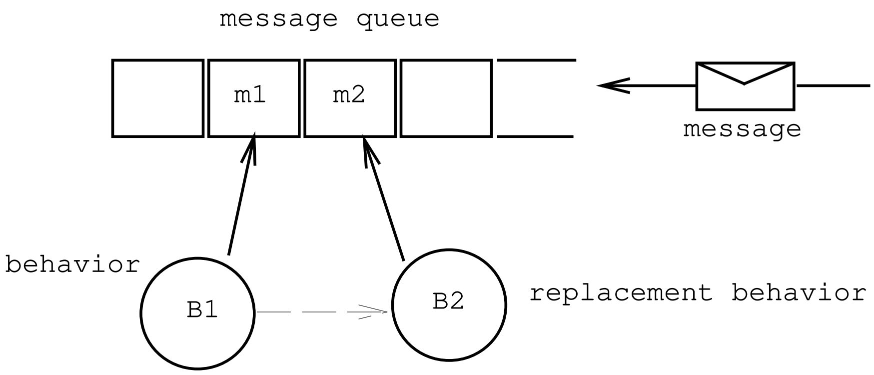

Figure 9: Behaviour replacement of an actor.

- *decentralised* schemes, such as *guards*, specify the synchronisation of the object at the application program level. As they are decentralised, each specification tends to be naturally associated and then integrated with a method.

There has been a big debate about the pros and cons of various schemes belonging to these two general categories. Important issues are: expressivity, reusability, provability, and efficiency. Several variations are described below, always with a focus on how they can be integrated with the object concepts. Note that most of the activity, synchronisation and communication models described in these sections have been implemented as various component libraries in a common framework for object-oriented concurrent programming named Actalk [Briot 1996]. This framework provides a relatively neutral platform to study, compare, and experiment with various models of activity (reactivity, body...), communication (synchronous, asynchronous...) and synchronization.

Path expressions. A first example of a centralised scheme is the concept of *path expressions* [Campbell and Habermann 1974] which specifies in a compact notation the possible sequences of invocations. The *Procol* language [van den Bos and Laffra 1991] is an example of integration of path expressions (called *protocols* in Procol) with objects.

The body revisited. Another example of a centralised scheme is the concept of *body* which has already been described above (Sect. 3.3.4). An important observation is that in complex cases the body may describe both application-specific behavior and the logic for accepting invocations. The missing distinction among these very different issues, their centralised handling in the body, plus its imperative nature, are the source of several problems when specialising object behaviors. In most cases the body needs to be rewritten from scratch. This general problem of reusing synchronisation specifications will be addressed in Section 3.6.1.

Behavior replacement. The Actor model of computation [Agha 1986] is based on three main concepts: active object, asynchronous message passing, and *behavior replacement*. This latter concept is both simple and very expressive. When created, an actor is composed of an address – to which is associated a mail queue buffering incoming messages – and an initial behavior. The behavior may be described as a set of variables/data and a set of methods, just as a standard object. The behavior will compute the first incoming message[9] and specify the *replacement behavior*, that is the behavior which will process the next message (Fig. 9). As one can imagine, once triggered by an incoming message, the replacement behavior will specify in turn its own replacement behavior, etc.

Note that as soon as the replacement behavior is specified, processing the next message may start. This implies intra-object concurrency. Vice-versa, as long as the replacement behavior is not specified, the computation of next message will not proceed. This implies synchronisation, and to be more precise, intra-object synchronisation.

Abstract states. The concept of replacement behavior encompasses the notion of a possibly changing behavior. If we combine that concept with the requirements for behavioral synchronisation described above (i.e., to delay the acceptance of a request, see Sect. 3.4.2[10]), and if we assume that the active object is serial, we obtain the concept of *abstract states*.

[9] In a similar way to objects, the message will select a method which will be evaluated in the environment of the behavior.

[10] This is realized by adding some degree of explicit message acceptance. Current behavior selects and computes, not the first pending message, but the first pending message *which matches one of its methods.*

```
class bounded_buffer : Actor {
  int_array buf[MAX];
  int in,out;
behavior:
  empty_buffer = {put()};
  full_buffer = {get()};
  partial_buffer = {get(),put()};
public:
  buffer() {
    in=0;
    out=0;
    become empty_buffer;
    }
void put(int item) {
    buf[in++]=item;
    in %= MAX;
    if (in==(out+1)%MAX)
      become full_buffer;
    else
      become partial_buffer;
    }
int get() {
    reply buf[out++];
    out %= MAX;
    if (in==out)
      become empty_buffer;
    else
      become partial_buffer;
    }
};
```

Figure 10: Bounded buffer in ACT++.

The idea[11] is the following: an object conforms to some abstract state representing a set of *enabled methods*. In the example of the bounded buffer, three abstract states are needed: `empty`, `full`, and `partial`. For instance, the abstract state `partial` is expressed as the union of `empty` and `full`, and consequently is the only one to enable both insertion and extraction methods. The corresponding program in *ACT++*, which is based on this idea, is shown in Fig. 10.

Guards. The notion of *guard* is a major example of decentralised synchronisation scheme. A guard is basically a boolean activation condition, which is associated with a procedure. The integration with objects is easy and natural: each method has one associated guard. Guards achieve a good integration because they do not require any synchronisation statements in the implementation of the object's operations. Activities are blocked or woken up implicitly. The price that has to be paid for this automatic scheme is performance; explicit operations such as signaling a monitor event or a semaphore are more efficient.

In the distributed programming language *Guide*, the guards are gathered in a central location of the class, called the control clause (keyword `CONTROL`, see Fig. 11, lower part).

Synchronisation 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 synchronisation. Referring to counters has the advantage of representation independence.

Note that, although integration of synchronisation schemes with the object model is usually straightforward, this integration may obstruct the reuse of synchronisation specifications (Sect. 3.6.1).

Locks. The concept of *lock* is one of the basic abstractions for synchronisation. It is very natural to

[11] See e.g., [Matsuoka and Yonezawa 1993] for a more detailed description.

```
CLASS Queue;
VAR length: Natural;
OPERATION remove: Item;
          WHEN length>0
          BEGIN ..... END;
.....
END Queue.
```

```
CLASS Queue IMPLEMENTS QueueType IS
.....
METHOD remove(OUT i: Item);
       BEGIN ..... END remove;
.....
CONTROL remove: completed(append) > completed(remove);
END Queue.
```

Figure 11: Guarded `remove` operation in Guide class `Queue`.

associate a lock to each object[12] in order for it to become a synchronised object. This is the approach followed by the *Java* programming language. As for Guide, Java is partially integrated in that it follows a model of synchronised object, but not a model of active object (objects and threads are kept separate). A private lock is implicitly associated with each Java object at its creation. Although this basic synchronisation abstraction is centralised in the object, the interface for the programmer is decentralised. Qualifying a method with the `synchronised` keyword indicates that this method is mutually exclusive with other qualified methods of the same class (or super-class). Actually the `synchronised` keyword can also be used for establishing arbitrary critical regions. Condition synchronisation is handled using events, another indication that Java favors a low degree of integration.

A language that allows for specifying reader/writer exclusion is *Distributed Eiffel* [Gunaseelan and LeBlanc 1992], designed as a modified Eiffel for programming distributed applications on top of the *Clouds* distributed operating system. An operation can be qualified as `ACCESSES` (or `MODIFIES`), meaning that it has to acquire a read lock (or a write lock) on the object before it can execute. If neither qualification is present, no lock is acquired.

Annotations. This approach is generalised in another Eiffel extension, CEiffel [Lohr 1993]: using *annotations* to the operations, a binary, symmetric *compatibility* relation among the operations of an object can be specified. If operation `op1` is declared compatible to operation `op2`, both can be executed in an overlapping fashion. Incompatible operations are mutually exclusive.

1. When a given sequential class without any annotations is used as a template for shared objects, these objects are serial by default, thus keeping their sequential semantics.
2. When a subclass extends the set of operations of the superclass, a new operation is incompatible with all the inherited operations, unless explicitly stated otherwise.

Figure 12 shows the example of readers and writers in Distributed Eiffel. Figure 13 shows the equivalent program in CEiffel.

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 possibly replication. Furthermore, self-containedness of objects (data plus procedures, plus possible internal activity) eases the 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

[12] or two locks, in order to distinguish between readers and writers methods.

```
CLASS Book
EXPORT number, read, write
FEATURE number: Integer IS
        DO ..... END;

        read: String ACCESSES
        DO ..... END;

        write(s: String) MODIFIES
        DO ..... END;
END -- Book
```

Figure 12: Reader/Writer exclusion using Distributed Eiffel.

```
CLASS Book
FEATURE number: Integer IS   --|| number --
        DO ..... END;

        read: String IS      --|| number, read --
        DO ..... END;

        write(s: String) IS  --|| number --
        DO ..... END;
END -- Book
```

Figure 13: Reader/Writer exclusion using CEiffel.

same or distinct processors is transparent to the programmer) as well as possible inaccessibility of an object/service.

3.5.1 Data-parallelism for distributed machines

If the concurrent activities of a program are to run in a truly parallel fashion, the program has to be mapped to a multiprocessor, a multicomputer or a computer network, giving rise to what is known as *functional* or *task parallelism*. For massive parallelism, however, there is more potential in *data parallelism* of the SPMD type (single program, multiple data) which is well-suited for distributed-memory architectures.[13]

ÉPÉE. ÉPÉE [Jézéquel 1993] follows the SPMD approach to data parallelism: large data aggregates of Eiffel code (such as matrices) are divided into fragments. The fragments are distributed, together with replicated code, over the CPUs of a multicomputer; each CPU operates on its data fragment, communicating with the other CPUs as necessary. ÉPÉE provides abstract structures which may be placed on several processors, without any addition to the Eiffel language.

The essentials of ÉPÉE are:

1. A data aggregate is an Eiffel object. Its interface is given by an Eiffel class. The class, however, describes the implementation of a fragment, *not* that of the complete aggregate.
2. Such a class for distributed aggregates must be designed as a subclass of a given non-distributed class, say, `Matrix`, and the class `DISTAGG`.
3. The original operations of `Matrix` have to be redefined. Their implementation has to be modified in such a way that update operations in the code are applied to the local fragment only. `DISTAGG` manages the required inter-fragment data exchange on remote read operations and provides various support functions such as fragment specific index mapping.

[13] Note that the SPMD data-parallelism model is opposed to the MIMD (multiple program/data) activation/control-parallelism model. Examples of the latter are languages based on the concept of *active objet* (Sect. 3.3). Indeed, objects represent duality (and unification) between data and procedures (potential activation).

4. There is no explicit process creation, nor any visible message passing. The fragments of a distributed object operate concurrently, each with its own thread of control. If each fragment is placed on a CPU of its own, invoking the object causes all the fragments to start operating in parallel.

Note that ÉPÉE, although integrating object and distribution through its concept of data aggregate, relies on class libraries for its implementation, without changing the underlying language (namely Eiffel).

. *Charm++* The *Charm++* language supports both MIMD and SPMD styles. Classes defined with the starting keywords `chare class` implement reactive active objects, similar to actors. But there exists a variant of the chare class concept called the *branched chare class* (instances of which are called branched chares). The code of a branched chare is replicated among the nodes of a computer network and each node works on one fragment of the object.

Although they look similar at first glance, there is a big difference between the object models of ÉPÉE and Charm++: the interface of a branched chare class reflects the fragmentation in that it describes the messages it can accept *from other fragments*, in addition to messages from other (outside) objects. Thus, although ÉPÉE has the advantage of hiding the explicit operations for inter-fragment message passing from the clients of an object, programming in Charm++ is less cumbersome because message passing is built into the language - as chare invocation. In summary, Charm++ achieves a better integration, at the cost of extending its underlying language (namely C++).

3.5.2 Accessibility and fault-recovery

In order to handle inaccessibility of objects, in the *Argus* distributed operating system [Liskov and Sheifler 1983] 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 with each invocation (synchronous invocation in Argus), to ensure atomicity properties. For instance, if the invocation fails (e.g., if the server object becomes inaccessible), the effects of the invocation are cancelled. The *Karos* distributed programming language [Guerraoui et al. 1992] extends the Argus approach by allowing the association of nested transactions with asynchronous invocations.

3.5.3 Migration

In order to improve the accessibility of objects, and also to support load balancing, some languages or systems provide mechanisms for object migration. In the Emerald distributed programming language [Jul et al. 1988] and the *COOL* generic run-time layer [Lea et al. 1993], the programmer may decide to migrate an object from one processor to another. In Emerald, this may be expressed at the message passing level as the migration of a parameter when the invocation is distant. This could be a permanent move (named "call-by move"), or temporary (named "call-by-visit"). The programmer may control (in terms of *attachments*) which other related objects should also migrate together. Similar concepts have also been included in a more recent, object-oriented system named *Dowl* [Achauer 1993], a distributed extension of the *Trellis/Owl* language [Moss and Kohler 1987].

3.5.4 Replication

As for migration, a first motivation of replication is to increase 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 *Electra* [Maffeis 1995] and *OGS* [Felber et al. 1998], the concept of remote invocation has been extended in such a way that invoking an object leads to the invocation of all its replicas and concurrent invocations are ordered along the same (total) order for all replicas. Support for transparent object replication in Emerald is described in [Black and Immel 1993].

3.6 Limitations of the Integrative Approach

The integrative approach attempts at unifying object mechanisms with concurrency and distribution mechanisms. This integration, however, leads to some conflicts, which we discuss below.

```
CLASS ClearableQueue INHERIT Queue

METHOD clear()
BEGIN  front := rear END clear

BODY DO IF    empty  THEN ANSWER(enq,clear)
        ELSIF full() THEN ANSWER(deq,clear)
        ELSE              ANSWER ANY FI OD YDOB
END ClearableQueue
```

Figure 14: Class ClearableQueue in POOL - Body needs to be completely rewritten.

3.6.1 Inheritance anomaly

Inheritance is one of the key mechanisms for achieving reuse of object-oriented programs. It is therefore natural to use inheritance to specialise synchronisation specifications associated with a class of objects. Unfortunately, experience shows that: (1) synchronisation is difficult to specify and, moreover, to reuse, because of the high interdependency between the synchronisation conditions for different methods; (2) various uses of inheritance (to inherit variables, methods, and synchronisations) may conflict with each other, as noted in [McHale 1994, Baquero et al. 1994]. In some cases, defining a new subclass, even only with one additional method, may force the redefinition of all synchronisation specifications. This limitation has been named the *inheritance anomaly phenomenon* [Matsuoka and Yonezawa 1993].

Specifications along centralised schemes (Sect. 3.4.3) turn out to be very difficult to reuse, and often must be completely redefined. This is for instance the case in POOL with the concept of body. The body is imperative rather than declarative, lacking the structure achieved by associating guards with operations. Figure 14 shows the definition in POOL of a subclass of class `Queue` (defined in Fig. 6), named `ExtendedQueue`. A new operation `clear` for emptying the queue has been added. Note that there is no way of reusing the body of the superclass; a complete redefinition is required. Recognising this, POOL requires that every class provide its own body. Thus, inheritance anomaly is the rule rather than the exception - except for the special case of empty bodies in both superclass(es) and subclass.

Decentralised schemes, being modular by essence, are better suited for selective specialisation. However, this fine-grained decomposition, at the level of each method, is only partially successful. This is because synchronisation specifications, even if decomposed for each method, still remain more or less interdependent. As for instance in the case of intra-object synchronisation with synchronisation counters, adding a new `write` method in a subclass may force redefinition of other guards, in order to account for the new mutual exclusion constraint.

The fact that intra-object concurrency (exclusion synchronisation) is implementation-dependent and therefore conceptually different from behavioral synchronisation (condition synchronisation) is recognised by some, though not all, languages. Guide employs counters not only for condition synchronisation, as mentioned above, but also for exclusion synchronisation. Unfortunately, this approach makes specifications more difficult to reuse. Figure 15 shows the definition in Guide of the same subclass `ClearableQueue` (see the original Guide definition of `Queue` in Fig. 11). Note that it does not suffice to add a new guard for `clear` to the control clause. Annoyingly, we have to redefine the `remove` guard although `remove` itself is not redefined.

There are various cases for inheritance anomaly and they have been carefully studied and classified in [Matsuoka and Yonezawa 1993]. Here are some recent directions proposed for minimising the problem:

1. Specifying and specialising independently behavioral synchronisation and intra-object synchronisation [Thomas 1992] as well as autonomy/asynchrony [Lohr 1993].
2. Specifying disabled methods rather than enabled methods, as disabled methods usually remain disabled in subclasses [Frolund 1992].
3. Allowing the programmer to select among several schemes [Matsuoka and Yonezawa 1993].
4. Offering a framework to help at designing, customising and combining between various synchronisation schemes [Briot 1996].
5. Instantiating abstract specifications, as an alternative to inheritance for reusing synchronisation specifications [McHale 1994]. Another example of high-level approach is the coordination patterns,

```
CLASS ClearableQueue INHERIT Queue
OPERATION clear;
          WHEN length>0
          BEGIN ..... END;
END ClearableQueue.
```

```
CLASS ClearableQueue SUBCLASS OF Queue IMPLEMENTS ClearableQueueType IS
METHOD clear;
        BEGIN ..... END clear;

CONTROL remove: completed(append) > completed(remove) + completed(clear);
        clear:  completed(append) > completed(remove) + completed(clear);
END ClearableQueue.
```

Figure 15: Class ClearableQueue in Guide - Major redefinition is necessary.

proposed by in [Frolund 1996], with a specific focus on inter-objects synchronisation. These abstractions are specified independently of the way an implementation can be (automatically) derived in order to help at reusing them.

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 specialisation may lead to incompatibility problems [Weihl 1989]. Broadly speaking, if objects use different transaction serialisation protocols (i.e., serialise the transactions along different orders), global executions of transactions may become inconsistent, i.e, non serialisable. A proposed approach to handle that problem is in defining local conditions, to be verified by objects, in order to ensure their compatibility [Weihl 1989, Guerraoui 1995].

3.6.3 Replication of objects and communications

The communication protocols which have been designed for fault-tolerant distributed computing (see Sect. 3.5.4) 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 [Mazouni et al. 1995], 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 Factorisation vs distribution

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

The concept of class variables, supported by several object-oriented programming languages (e.g., Smalltalk), 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 methodologies tend 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.

A 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 [Lieberman 1987]. Intuitively, an object which may not understand a message will then delegate it (i.e., forward it[14]) 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, and hence fits well with a distributed implementation. Meanwhile, the delegation mechanism needs some non trivial synchronisation 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 [Briot and Yonezawa 1994].

4 The Reflective Approach

4.1 Combining Flexibility and Transparency

As discussed earlier, the library approach helps at structuring concurrent and distributed programming concepts and mechanisms, thanks to encapsulation, genericity, class, and inheritance concepts. The integrative approach minimises 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. In other words, 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 models. It would thus be interesting to keep the unification and simplification advantages of the integrative approach, while retaining the flexibility of the library approach.

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

Therefore, and as opposed to what one may think at first, the *library* 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 a glue.

4.2 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. Static representation characteristics, as well as dynamic execution characteristics of application programs are represented by a program which represents the default computational behavior (interpreter, compiler, execution monitor...). Such a description/control program is called a *meta-program*. Specialising such programs allows one to customise the execution of the application program, by possibly changing data representations, 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 mod-

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

ularity, readability and reusability of programs. Reflection also provides a methodology to open up and make adaptable, through a *meta-interface*[15], implementation decisions and resource management, which are often hard-wired and fixed, or delegated to the underlying operating system.

In summary, reflection helps at integrating protocol libraries intimately with a programming language or system, thus providing the interfacing framework (the glue) between the library approach and the integrative approach.

4.3 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 organise the control of the behavior of an object-oriented computational system (its meta-interface) through a set of objects. This organisation is named a *Meta-Object Protocol (MOP)* [Kiczales et al. 1991], and its components are called *meta-objects* [Maes 1987], as meta-programs are represented by objects. They may represent various characteristics of the execution context such as: representation, implementation, execution, communication and location. Specialising meta-objects may extend and modify the execution context of some specific objects of the application program.

Reflection may also help at expressing and controlling resource management, not only at the level of an individual object, but also at a broader level such as: scheduler, processor, name space, object groups, such resources being also represented by meta-objects. This helps with a very fine-grained control (e.g., for scheduling and load balancing) with the whole expressive power of a full programming language [Okamura and Ishikawa 1994], as opposed to some global and fixed algorithm (which is usually optimised for a specific kind of application or an average case).

4.4 Examples of Meta Object Architectures

4.4.1 Smalltalk

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 category. In addition to the meta-representation of the program structures and mechanisms as first class objects (Sect. 2.2), very powerful reflective mechanisms offer some control over program execution. Examples are: redefinition of error handling messages, reference to current context, reference swap, and changing the class of an object. Such facilities actually alleviates building and integrating various platforms for concurrent and distributed programming, such as Coda, Actalk, and GARF.

4.4.2 Coda

The *CodA* architecture [McAffer 1995] is a representative example of a general object-based reflective architecture (i.e., a MOP) based on *meta-components*[16]. CodA considers by default seven meta-components, associated with each object (Fig. 16), corresponding to: *message sending*, *receiving*, *buffering*, *selection*, *method lookup*, *execution*, and *state access*. An object with default meta-components behaves as a standard (sequential and passive) object (to be more precise, as a standard Smalltalk object, as CodA is currently implemented in Smalltalk). Attaching specific meta-components allows to selectively change 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, in the case of the CodA system, as well as for almost all other examples of reflective systems described below, the basic programming model is *integrative*, while reflection enables the customisation of concurrency and distribution aspects and protocols by specialising *libraries* of meta-components.

[15]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 the concept of *open implementation* [Kiczales 1994].

[16]Note that meta-components are indeed meta-objects. In the following, we will rather use the term *meta-component* in order to emphasise 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*.

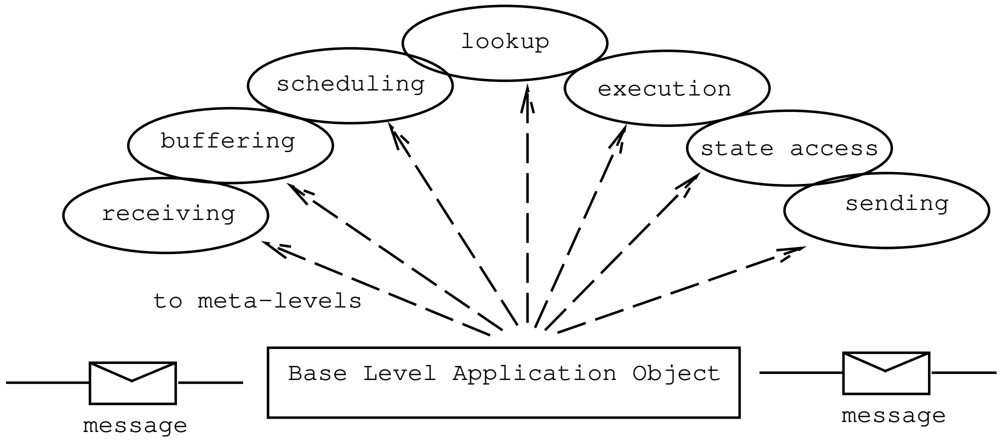

Figure 16: Meta-components in CodA.

In order to introduce concurrency for a given object (by making it into an *active* object, taking an integrated approach), two meta-components are specialised. The specialised *message buffering* component[17] is a queue which will buffer incoming messages. The specialised *execution* component associates an independent activity (thread) to the object. This thread executes an endless loop for selecting and processing the next message from the *buffering* component.

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 specialised *message receiving* component, which marshals the message into a stream of bytes and sends it through the network to the actual remote object. The remote object has another specialised *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 specialised by a *marshaling descriptor* supplied by the *marshaling* component.

Migration is introduced by a new meta-component which describes the form and the policy (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 [Black et al. 1987]), with attachments, may be specialised through the *marshaling descriptors* supplied by the corresponding component. Also one may specialise 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.

4.4.3 Actalk and GARF

Reflective architectures may be more specialised and may offer a more reduced (and abstract) set of meta-components. Examples are the Actalk and GARF platforms, where a smaller amount of meta-components suffices to express a large variety of schemes and application problems.

The *Actalk* platform [Briot 1989, Briot 1996] helps at experimenting with various synchronisation and communication models for a given program, by changing and specialising various models/components of: (1) *activity* (e.g., implicit or explicit acceptance of requests, intra-objet concurrency) and *synchronisation* (e.g., abstract behaviors, guards), (2) *communication* (synchronous, asynchronous…), and (3) *invocation* (e.g., time stamp, priority). The *GARF* platform [Garbinato et al. 1994] for distributed and fault-tolerant programming, offers a variety of mechanisms along two dimensions/components: (1) object control (e.g., persistence and replication) and (2) communication (e.g., asynchronous and multicast).

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

4.5 Further Application Examples

Other examples of representative reflective architectures and their applications are briefly mentioned below.

4.5.1 Dynamic installation and protocol composition

The general MAUD methodology [Agha et al. 1993] focuses on fault tolerance protocols, such as server replication and checkpointing, Its specificity is in offering a framework for *dynamic installation* and *composition* of specialised 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 with objects, but also with other meta-components (which are first-class objects), enables the layered composition of protocols. A higher-level layer for specification of dependability protocols (called DIL [Sturman and Agha 1994]) has been designed and makes use of the underlying MAUD reflective architecture.

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. Nevertheless, the decision to migrate an object is an important issue which often remains with the programmer's responsibility (e.g., in Emerald [Black et al. 1987]). It may be interesting to semi-automate such a decision, along various considerations such as processor load and 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 [Okamura and Ishikawa 1994].

4.5.3 Customising system policies

The Apertos distributed operating system [Yokote 1992] 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 a library approach (as for Choices), reflection brings the (possibly dynamic) customisation of the system towards application requirements. For instance one may easily specialise 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 resource abstractions. This helps with 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[18] 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 [Barga and Pu 1994]. These reflective primitives are then used to implement various extended transaction model, such as: split/join, cooperative groups.

4.5.5 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 [Aksit et al. 1993]. 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.

4.5.6 Generic run-time as a dual approach

The boundary 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

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

for instance the COOL layer for the Chorus operating system [Lea et al. 1993]. These generic run time layers are designed for use by various programming languages, thanks to some upcalls which delegate specific representation decisions to the programming language.

5 PERSPECTIVES

5.1 The Library Approach

The library approach aims at increasing the flexibility, yet reducing the complexity, of concurrent and distributed computing systems by structuring them as class libraries. Each aspect or service is represented by an object. Such modularity and abstraction features are very important because concurrent 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.

Although progress is made towards that direction, it is still too early to exhibit a standard class library for concurrent and distributed programming. We need both a good knowledge of the minimal mechanisms required and a consensus on a set of such mechanisms, involving different technical communities, notably: programming languages, operating systems, distributed systems and databases. The fact that the *semaphore* and *monitor* abstractions became standard primitives for synchronisation may indicate that other abstractions for concurrent and distributed programming could also be identified and adopted.

5.2 The Integrative Approach

The integrative approach is very appealing because of the merging it achieves between concepts of object-oriented programming and those of concurrent and distributed programming. It thus provides a minimal number of concepts and presents a single conceptual framework to the programmer. Nevertheless, this approach unfortunately suffers from limitations in some aspects of the integration (e.g., inheritance anomaly, duplication anomaly).

Another potential weakness is that a too systematic unification/integration may lead to a 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. A last important limitation is a *legacy* problem, that is the possible difficulty with reusing standard sequential programs. A straightforward solution is the encapsulation of sequential programs into active objects. However, a coexistence between active objects and passive objects requires specific methodological rules for handling this heterogeneity [Caromel 1993].

5.3 The Reflective Approach

Reflection provides a general framework for the customisation of concurrency and distribution aspects and protocols, by specialising 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 concurrent and distributed programming (although we believe that CodA [McAffer 1995] is a promising step in that direction). Note that there is currently a big effort in designing reflective architectures (MOPs) for the C++ programming language. The goal is to offer some generic framework in order to be able to express various models and protocols of parallel and distributed programming. Two significant examples of such effort are: OpenC++ [Chiba 1995] and C++// [Caromel et al. 1996].

Meanwhile, we still need more experience in the practical use of reflection, to be able to find good tradeoffs between the flexibility required, the architectural complexity, and the resulting efficiency. One possible (and currently justified) complaint is about the actual relative complexity of reflective architectures. Nevertheless, and independent of the required cultural change, we believe that this is the price to be paid for the increased, albeit disciplined, flexibility that they offer. Another significant current limitation concerns efficiency, due to the extra indirections and interpretations. Two alternative directions are: (1) to reduce the scope of reflection at compile-time – it is well witnessed by the evolution of the initial reflective OpenC++ architecture into a compile-time reflective architecture, thus getting closer to meta-compilers [Chiba 1995] –, or (2) to use program transformation techniques, notably partial evaluation (also called program specialisation), to minimise meta-interpretation overheads [Masuhara et al. 1995].

5.4 Integrating the Approaches

As already pointed out in Section 1.2, the *library*, *integrative* and *reflective* approaches are not in conflict, but rather complementary. The complementary nature of these approaches extends to their relationship to language: the library approach does not change the underlying language; the library approach either defines a new language or adds new concepts to the language; and the reflective approach requires the use of a specific type of language.

Among the examples of languages and systems given in the paper, some have been built following more than one approach. This is the case for instance for the ÉPÉE [Jézéquel 1993] parallel system (Sect. 3.5.1), which on the one hand is based on the integration of object with distribution, and on the other hand is implemented with libraries. Other examples are Actalk [Briot 1989] and GARF [Garbinato et al. 1994] (Sect. 2.2), which offer libraries of abstractions for concurrent and distributed programming that may be transparently applied to programs thanks to the reflective facilities of Smalltalk.

We believe that future developments of object-based concurrent and distributed systems will integrate aspects of the three approaches. A very good example is the current development around the Common Object Request Broker Architecture (CORBA) of the OMG [Mowbray and Zahavi 1995]. CORBA *integrates* object and distribution concepts through an object request broker (which provides remote communication facilities). In that sense, CORBA follows the *integrative* approach. CORBA also specifies a set of services to support more advanced distributed features such as transactions. The CORBA object transaction service (named *OTS*) is specified and implemented in the form of a class library of distributed protocols, such as *locking* and *atomic commitment*. In that sense, CORBA follows the *library* approach. Finally, most CORBA implementations provide facilities for message reification (messages can be considered as first class entities), and hence support customisation of concurrency and distribution protocols. In that sense, CORBA implementations follow (to some extent) the *reflective* approach.

6 Summary

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

The *library* approach (library-based approach) helps at structuring concurrent 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 concurrent and distributed architecture is are represented by unrelated sets of concepts and objects. The library approach can be viewed as a bottom-up approach and is directed towards system builders.

The *integrative* approach minimises the amount of concepts to be mastered by the programmer and makes mechanisms more transparent, by providing a unified concurrent and distributed high level object model, however, 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.

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 library 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 customisations and thus change the execution context with minimal changes on the application programs.

The reflective approach contributes to blurring the distinction between programming language, operating system, and database, and supports the development, adaptation and optimisation of a minimal computing system which is dynamically extensible. This however does not free us from the necessity of a good basic design and finding a good set of foundational abstractions.

References

[Achauer 1993] Achauer, B., 1993. The Dowl distributed object-oriented language. *Comm. ACM 36(9).*

[Ada 1983] Ada 1983. *The Programming Language Ada Reference Manual.* LNCS 155, Springer-Verlag.

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

[Agha et al. 1989] Agha, G.A., Wegner, P., Yonezawa, A., Eds., 1989. *Proc. ACM Sigplan Workshop on Object-Based Concurrent Programming. ACM Sigplan Not. 24(4).*

[Agha et al. 1991] Agha, G.A., Hewitt, C., Wegner, P., Yonezawa, A., Eds., 1991. *Proc. OOPSLA/ECOOP '90 Workshop on Object-Based Concurrent Programming. ACM OOPS Messenger 2(2).*

[Agha et al. 1993] Agha, G.A., Wegner, P., Yonezawa, A., Eds., 1993. *Research Directions in Concurrent Object-Oriented Programming,* M.I.T. Press.

[Agha et al. 1993] Agha, G.A., Frølund, S., Panwar, R., Sturman, D., 1993. A linguistic framework for dynamic composition of dependability protocols. *Dependable Computing for Critical Applications III (DCCA-3)*, IFIP Transactions, Elsevier, 197–207.

[Aksit et al. 1993] Aksit, M., Wakita, K., Bosch, J., Bergmans, L. Yonezawa, A., 1994a. Abstracting object interactions using composition filters. In [Guerraoui et al. 94], 152–184.

[America 1987] America, P.H.M., 1987. Pool-T: a parallel object-oriented language. In [Yonezawa/Tokoro 87].

[America 1988] America, P.H.M., 1988. Definition of Pool2, a parallel object-oriented language. ESPRIT project 415-A, report 364. Philips Research Laboratories.

[America 1989] America, P.H.M., 1989. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing 1,* 366-411

[America and van der Linden 1990] America, P.H.M., van der Linden, F., 1990. A parallel object-oriented language with inheritance and subtyping. *Proc. OOPSLA/ECOOP '90, ACM Sigplan Not. 25(10).*

[Andrews et al. 1993] Andrews, G.R., Olsson, R.A., 1993. *The SR Programming Language,* Benjamin/Cummings.

[Balter et al. 1994] Balter, R., Lacourte, S., Riveill, M., 1994. The Guide language. *The Computer Journal 37(6)*, 519–530.

[Baquero et al. 1994] Baquero, C., Oliveira, R. Moura, F., 1995. Integration of concurrency control in a language with subtyping and subclassing. *USENIX COOTS Conference (COOTS'95)*, Monterey, CA.

[Barga and Pu 1994] Barga, R., Pu, C., 1995. A practical and modular implementation of extended transaction models. Technical Report 95-004, CSE, Oregon Graduate Institute of Science & Technology, Portland, OR.

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

[Bershad et al. 1988] Bershad, B.N., Lazowska, E.D., Levy, H.M., 1988. PRESTO: a system for object-oriented parallel programming. *Software - Practice and Experience 18(8),* 713-732.

[Bezivin 1987] Bézivin, J., 1987. Some Experiments in Object-Oriented Simulation. *ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87),* 394–405.

[Birtwistle et al. 1973] Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B., Nygaard, K., 1973. *Simula Begin*, Petrocelli Charter.

[Black et al. 1987] Black, A.P., Hutchinson, N., Jul, E., Levy, H., Carter, L., 1987. Distribution and abstract types in Emerald. *IEEE Trans. on Software Engineering 13(1).*

[Black 1991] Black, A.P., 1991. Understanding transactions in the operating system context. *Operating Systems Review 25*, 73–77.

[Black and Immel 1993] Black, A.P., Immel, M.P., 1993. Encapsulating Plurality. *Proc. European Conf. on Object-Oriented Programming (ECOOP'93)*, LNCS 707, Springer-Verlag, 57–79.

[van den Bos and Laffra 1991] van den Bos, J., Laffra, C., 1991. Procol - a concurrent object-oriented language with protocols, delegation and constraints. *Acta Informatica 28,* 511-538.

[Brandt and Lehrmann Madsen 1994] Brandt, S., Lehrmann Madsen, O., 1994. Object-Oriented Distributed Programming in BETA. In [Guerraoui et al. 94], 185–212.

[Briot and Yonezawa 1994] Briot, J.-P., Yonezawa, A., 1987. Inheritance and synchronisation in concurrent OOP. *Proc. European Conf. on Object-Oriented Programming (ECOOP'87)*, LNCS 276, Springer-Verlag, 32–40.

[Briot 1989] Briot, J.-P., 1989. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. *Proc. European Conf. on Object-Oriented Programming (ECOOP'89)*, Cambridge University Press, 109–129.

[Briot et al. 1995] Briot, J.-P., Geib, J.-M., Yonezawa, A., Eds., 1995. *Proc. France-Japan Workshop on Object-Based Parallel and Distributed Computation,* LNCS 1107, Springer-Verlag.

[Briot 1996] Briot, J.-P., 1996. An experiment in classification and specialization of synchronisation schemes. *Proc. 2. Int. Symp. on Object Technologies for Advanced Software (ISOTAS'96)*, LNCS, Springer-Verlag.

[Campbell and Habermann 1974] Campbell, R.H., Habermann, A.N., 1974. The specification of process synchronisation by path expressions. In Gelenbe, E., Kaiser, C., Eds.: *Operating Systems,* LNCS 16, Springer-Verlag.

[Campbell et al. 1993] Campbell, R., Islam, N., Raila, D., Madany, P., 1993. Designing and implementing Choices: an object-oriented system in C++. *Comm. ACM 36(9)*, 117–126.

[Caromel 1989] Caromel, D., 1989. Service, asynchrony and wait-by-necessity. *J. of Object-Oriented Programming 2(4).*

[Caromel 1990] Caromel, D., 1990. Concurrency and reusability: from sequential to parallel. *J. of Object-Oriented Programming 3(3).*

[Caromel 1993] Caromel, D., 1993. Towards a method of object-oriented concurrent programming. *Comm. ACM 36(9),* 90-102.

[Caromel et al. 1996] Caromel, D., Belloncle, F., Roudier, Y., 1996. The C++// System. [Wilson and Lu 1996], pages 257–296.

[Chiba 1995] Chiba, S., 1995. A Metaobject Protocol for C++. *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), Special Issue of Sigplan Notices*, Vol. 30, n^o 10, October 1995, pages 285–299.

[Felber et al. 1998] Felber, P., Guerraoui, R., Schiper, A., 1993. The Implementation of a CORBA Object Group Service. *Theory and Practice of Object Systems, John Wiley and Sons, Inc. Special issue on High Availability in CORBA, 4(2),* 93-106.

[Finke et al. 1995] Finke, S., Jahn, P., Langmack, O., Löhr, K.-P., Piens, I., Wolff, Th., 1993. Distribution and inheritance in the HERON approach to heterogeneous computing. *Proc. 13. Int. Conf. on Distributed Computing Systems,* IEEE.

[Frolund 1992] Frølund, S., 1992. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. *Proc. European Conf. on Object-Oriented Programming (ECOOP '92),* LNCS 615, Springer-Verlag.

[Frolund 1996] Frølund, S., 1996. *Coordinating Distributed Objects,* MIT Press.

[Garbinato et al. 1994] Garbinato, B., Guerraoui, R. and Mazouni, K.R., Distributed Programming in GARF, [Guerraoui et al. 1993], pages 225–239.

[Garbinato et al. 1995] Garbinato, B., Guerraoui, R. and Mazouni, K.R., Implementation of the GARF Replicated Objects Platform, *Distributed Systems Engineering Journal*, February 1995, pages 14–27.

[Garbinato et al. 1996] Garbinato, B., Felber, P., and Guerraoui, R., Protocol classes for designing reliable designing reliable distributed environments, *European Conference on Object Oriented Programming (ECOOP'96)*, edited by P. Cointe, LNCS, n^o 1098, Springer-Verlag, June 1996, pages 316–343.

[Garbinato and Guerraoui 1997] Garbinato, B., and Guerraoui, R., Using the strategy design pattern to compose reliable distributed protocols, *Usenix Conference on Object-Oriented Technologies and Systems (COOTS'97)*, edited by S.Vinoski, Usenix, June 1997.

[Goldberg and Robson 1989] GOLDBERG, A., ROBSON, D., 1989. *Smalltalk-80. The Language*, Addison-Wesley.

[Guerraoui et al. 1992] GUERRAOUI, R., CAPOBIANCHI, R., LANUSSE, A., ROUX, P. 1992. Nesting actions through asynchronous message passing: the ACS protocol. *Proc. European Conf. on Object-Oriented Programming (ECOOP'92)*, LNCS 615, Springer-Verlag, 170–184.

[Guerraoui et al. 1993] GUERRAOUI, R., NIERSTRASZ, O., RIVEILL, M., EDS., 1994. *Proc. ECOOP '93 Workshop on Object-Based Distributed Programming*, LNCS 791, Springer-Verlag.

[Guerraoui 1995] GUERRAOUI, R., 1995. Modular atomic objects. *Theory and Practice of Object Systems 1(2)*, 89–99.

[Guerraoui and Schiper 1995] GUERRAOUI, R. AND SCHIPER, A., 1995. Transactional model vs Virtual Synchrony model: bridging the gap. *Theory and Practice in Distributed Systems,* Springer Verlag (LNCS 938), 1995, 121-132.

[Guerraoui and Schiper 1996] GUERRAOUI, R. AND SCHIPER, A., 1996. Consensus Service: A Modular Approach for Building Agreement Protocols in Distributed Systems. IEEE Symposium on Fault-Tolerant Computing, pages 168-177, June 1996.

[Gunaseelan and LeBlanc 1992] GUNASEELAN, L., LEBLANC, R.J., 1992. Distributed Eiffel: a language for programming multi-granular distributed objects. *Proc. 4. Int. Conf. on Computer Languages,* IEEE.

[Halstead 1985] HALSTEAD, R.H., 1985. Multilisp: a language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems 7(4).*

[Jézéquel 1993] JÉZÉQUEL, J.-M., 1993. EPEE: an Eiffel environment to program distributed-memory parallel computers. *J. of Object-Oriented Programming 6(2).*

[Jul et al. 1988] JUL, E., LEVY, H.M., HUTCHINSON, N.C., BLACK, A.P., 1988. Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems 6(1).*

[Kafura and Lee 1990] KAFURA, D.G., LEE, K.H., 1990. ACT++: building a concurrent C++ with actors. *J. of Object-Oriented Programming 3(1).*

[Kalé and Krishnan 1993] KALÉ, L.V., KRISHNAN, S., 1993. Charm++: a portable concurrent object-oriented system based on C++. *Proc. ACM Conf. on Object-Oriented Systems, Languages and Applications (OOPSLA '93), ACM Sigplan Not. 28.*

[Karaorman and Bruno 1993] KARAORMAN, M., BRUNO, J., 1993. Introducing concurrency to a sequential language. *Comm. ACM 36(9)*, 103–116.

[Kay 1969] KAY, A., 1969. The Reactive Engine. PhD thesis, University of Utah, 1969.

[Kiczales et al. 1991] KICZALES, G., DES RIVIÈRES, J., BOBROW, D., 1991. *The Art of the Meta-Object Protocol*, MIT Press.

[Kiczales 1994] KICZALES, G., ED., 1994. Foil for the Workshop On Open Implementation. `http://www.parc.xerox.com/PARC/spl/eca/oi /workshop-94/foil/main.html`.

[Lea et al. 1993] LEA, R., JACQUEMOT, C., PILLEVESSE, E., 1993. COOL: System support for distributed programming. *Comm. ACM 36(9)*, 37–47.

[Lea 1997] LEA, D., 1997. *Concurrent Programming in Java,* Addison-Wesley.

[Lehrmann Madsen et al. 1993] LEHRMANN MADSEN, O., MØLLER-PEDERSEN, B., NYGAARD, K., 1993. *Object-Oriented Programming in the BETA Programming Language,* Addison-Wesley.

[Lieberman 1987] LIEBERMAN, H., 1987. Concurrent object-oriented programming in Act 1. In [Yonezawa/Tokoro 87], 9–36.

[Liskov and Sheifler 1983] LISKOV, B., SHEIFLER, R., 1983. Guardians and actions: linguistic support for robust, distributed programs. *ACM Trans. on Programming Languages and Systems 5(3).*

[Lohr 1992] LÖHR, K.-P., 1992. Concurrency Annotations. *Proc. ACM Conf. on Object-Oriented Systems, Languages and Applications (OOPSLA '92), ACM Sigplan Not. 27(10).*

[Lohr 1993] LÖHR, K.-P., 1993. Concurrency annotations for reusable software. *Comm. ACM 36(9),* 81-89.

[Lopes and Lieberherr 1994] Lopes, C.V., Lieberherr, K.J., 1994. Abstracting process-to-function relations in concurrent object-oriented applications. *Proc. European Conf. on Object-Oriented Programming (ECOOP'94)*, LNCS 821, Springer-Verlag, 81–99.

[Maes 1987] Maes, P., 1987. Concepts and experiments in computational reflection. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), ACM Sigplan Not. 22(12)*, 147–155.

[Maffeis 1995] Maffeis, S., 1995. *Run-Time Support for Object-Oriented Distributed Programming,* PhD dissertation, Universität Zürich.

[Masuhara et al. 1995] Masuhara, H., Matsuoka, S., Asai, K., Yonezawa, A., 1995. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95), ACM Sigplan Not. 30(10)*, 300–315.

[Matsuoka and Yonezawa 1993] Matsuoka, S., Yonezawa, A., 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In [Agha et al. 93], 107-150.

[Mazouni et al. 1995] Mazouni, K., Garbinato, B., Guerraoui, R., 1995. Building reliable client-server software using actively replicated objects. *Proc. Conf. on Technologies of Object-Oriented Systems, Languages and Applications (TOOLS Europe '95)*, Prentice-Hall, 37–53.

[McAffer 1995] McAffer, J., 1995. Meta-level programming with CodA. *Proc. European Conf. on Object-Oriented Programming (ECOOP'95),* LNCS 952, Springer-Verlag, 190–214.

[McHale 1994] McHale, C., 1994. *Synchronisation in Concurrent, Object-Oriented Languages: Expressive Power, Genericity and Inheritance,* PhD thesis, Dept. of Computer Science, Trinity College, Dublin.

[Meyer 1991] Meyer, B., 1991. *Eiffel: The Language.* Prentice-Hall.

[Meyer 1993] Meyer, B., 1993. Systematic concurrent object-oriented programming. *Comm. ACM 36(9)*, 56–80.

[Moss and Kohler 1987] Moss, J.E.B., Kohler, W.H., 1987. Concurrency features for the Trellis/Owl language. *Proc. European Conf. on Object-Oriented Programming (ECOOP '87),* LNCS 276, Springer-Verlag.

[Mowbray and Zahavi 1995] Mowbray, T.J., and Zahavi, R., 1995 *The Essential CORBA: System Integration Using Distributed Objects,* John Wiley & Sons and The Object Management Group.

[Nicol et al. 1993] Nicol, J., Wilkes, T., Manola, F., 1993. Object-orientation in heterogeneous distributed computing systems. *IEEE Computer 26(6)*, 57–67.

[Okamura and Ishikawa 1994] Okamura, H., Ishikawa, Y., 1994. Object location control using meta-level programming. *Proc. European Conf. on Object-Oriented Programming (ECOOP '94)*, LNCS 821, Springer-Verlag, 299–319.

[OMG 1995] OMG 1995. *The Common Object Request Broker: Architecture and Specification (Revision 2.0).* Object Management Group, Framingham, Mass.

[OSF 1994] OSF 1994. *DCE Application Development Guide (Revision 1.0.2).* Open Software Foundation, Cambridge, Mass.

[Papathomas 1989] Papathomas, M., 1989. Concurrency issues in object-oriented programming languages. In Tsichritzis, D.C., Ed.: *Object-Oriented Development,* Centre Universitaire d'Informatique, Université de Genève.

[Papathomas 1995] Papathomas, M., 1995. Concurrency in object-oriented programming languages. In Nierstrasz, O., Tsichritzis, D., Eds.: *Object-Oriented Software Composition,* Prentice-Hall, 31-68.

[Parrington and Shrivastava 1988] Parrington, G.D., Shrivastava, S.K., 1988. Implementing concurrency control in reliable distributed object-oriented systems. *Proc. European Conf. on Object-Oriented Programming (ECOOP'88)*, LNCS 322, Springer-Verlag, 234–249.

[Rozier 1992] Rozier, M., 1992. Chorus. *Usenix Int. Conf. on Micro-Kernels and Other Kernel Architectures*, 27–28.

[Schill and Mock 1993] Schill, A., Mock, M. 1993. DC++: Distributed object-oriented system support on top of OSF DCE. *Distributed Systems Engineering 1(2)*, 112-125.

[Schmid 1995] SCHMID, D.C., 1995. An OO encapsulation of lightweight OS concurrency mechanisms in the ACE toolkit. TR WUCS-95-31, Dept. of Computer Science, Washington University, St. Louis.

[Schneider 1985] SCHNEIDER, F. 1985. Paradigms for distributed programs. Distributed Systems-Methods and Tools for Specification, Lecture Notes in Computer Science, Vol. 190, Springer Verlag, 1985, 343-430.

[Schröder-Preikschat 1994] SCHRÖDER-PREIKSCHAT, W., 1994. *The Logical Design of Parallel Operating Systems*, Prentice-Hall.

[Sheffler 1996] SHEFFLER, TH.J., 1996. The Amelia Vector Template Library. In [Wilson and Lu 1996], 43-90.

[Skjellum et al. 1996] SKJELLUM, A., LU, Z., BANGALORE, P.V., DOSS, N., 1996. MPI++. In [Wilson and Lu 1996], 465-506.

[Stroustrup 1993] STROUSTRUP, B., 1993. *The C++ Programming Language*. Addison-Wesley.

[Sturman and Agha 1994] STURMAN, D., AGHA, G., 1994. A Protocol Description Language for Customizing Failure Semantics. *13th International Symposium on Reliable Distributed Systems*, 148–157.

[Sun 1995] SUN 1995. *C++4.1 Library Reference Manual, Section 2.* Part No. 802-3045-10, Nov. 1995, Sun Microsystems Inc.

[Thomas 1992] THOMAS, L., 1992. Extensibility and reuse of object-oriented synchronisation components. *Proc. Int. Conf. on Parallel Languages and Environments (PARLE '92)*, LNCS 605, Springer-Verlag, 261–275.

[Tokoro et al. 1992] TOKORO, M., NIERSTRASZ, O.M., WEGNER, P., EDS., 1992. *Proc. ECOOP '91 Workshop on Object-Based Concurrent Computing,* LNCS 612, Springer-Verlag.

[Wegner 1990] WEGNER, P., 1990. Concepts and paradigms of object-oriented programming. *ACM OOPS Manager 1(1).*

[Weihl 1989] WEIHL, W., 1989. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. on Programming Languages and Systems 11(2).*

[Wilson and Lu 1996] WILSON, G.V., LU, P., EDS., 1996. *Parallel Programming Using C++,* MIT Press.

[Yokote and Tokoro 1987] YOKOTE, Y., TOKORO, M., 1987. Experience and evolution of Concurrent Smalltalk. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications OOPSLA '87), ACM Sigplan Not. 22(12).*

[Yokote 1992] YOKOTE, Y., 1992. The Apertos reflective operating system: the concept and its implementation. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92), ACM Sigplan Not. 27(10)*, 414–434.

[Yonezawa et al. 1986] YONEZAWA, A., BRIOT, J.-P., SHIBAYAMA, E., 1986. Object-oriented concurrent programming in ABCL/1. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86), ACM Sigplan Not. 21(11).*

[Yonezawa and Tokoro 1987] YONEZAWA, A., TOKORO, M., EDS., 1987. *Object-Oriented Concurrent Programming*, Computer Systems Series, MIT Press.

[Yonezawa 1993] YONEZAWA, A., MATSUOKA, S., YASUGI, M., TAURA, K., 1993. Implementing concurrent object-oriented languages on multicomputers. *IEEE Parallel and Distributed Technology,* May 1993.