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; email: ⟨Jean-Pierre.Briot@lip6.fr⟩.

RACHID GUERRAOUI

Département d'Informatique, École Polytechnique Fédérale de Lausanne, CH-1015, Lausanne, Switzerland; email: ⟨rachid.guerraoui@epfl.ch⟩.

AND

KLAUS-PETER LOHR

Institut für Informatik, Freie Universität Berlin, D-14195 Berlin, Germany; email: ⟨lohr@inf.fu-berlin.de⟩.

This paper aims at discussing and classifying the various ways in which the object paradigm is used in concurrent and distributed contexts. We distinguish among 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 of 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 and point out their complementary levels and goals.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures; C.2.4 [**Computer–Communication Networks**]: Distributed Systems; D.1 [**Software**]: Programming Techniques; D.3.2 [**Programming Languages**]: Language Classifications; D.4.1 [**Operating Systems**]: Process Management; D.4.4 [**Operating Systems**]: Communications Management; D.4.5 [**Operating Systems**]: Reliability; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence; H.2.4 [**Database Management**]: Systems

General Terms: Languages, Reliability, Performance

Additional Key Words and Phrases: Concurrency, distribution, integration, libraries, message passing, object, reflection

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 object-based programming languages do have some concurrent or distributed extension(s), and almost every

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1998 ACM 0360-0300/98/0300–0291 $5.00

CONTENTS

1. INTRODUCTION
 1.1 A Coarse Classification in Three Approaches
 1.2 Complementarity of the Approaches
 1.3 Concurrency and Distribution
 1.4 Previous Work
2. THE LIBRARY APPROACH
 2.1 Modularity and Structuring Needs
 2.2 Smalltalk Libraries
 2.3 C++ Libraries
 2.4 In Search of Standard Abstractions
3. THE INTEGRATIVE APPROACH
 3.1 Unification Needs
 3.2 Dimensions of Integration
 3.3 Active Objects
 3.4 Synchronised Objects
 3.5 Distributed Objects
 3.6 Limitations of the Integrative Approach
4. THE REFLECTIVE APPROACH
 4.1 Combining Flexibility and Transparency
 4.2 Reflection
 4.3 Reflection and Objects
 4.4 Examples of Meta Object Protocols (MOPs)
 4.5 Examples of Applications
 4.6 Other Examples of Reflective Architectures
 4.7 Related Models
5. PERSPECTIVES
 5.1 The Library Approach
 5.2 The Integrative Approach
 5.3 The Reflective Approach
 5.4 Integrating the Approaches
6. SUMMARY

new architectural development in the distributed system community is, to some extent, object-based. For instance, both the *Open Distributed Processing* (ODP) and the *Object Management Group* (OMG), recent standardization initiatives for heterogeneous distributed computing, are based on object concepts [Nicol et al. 1993; OMG 1995; Mowbray et al. 1995.

As a result, many object-based concurrent, parallel, or distributed models, languages, or system architectures have been proposed and described in the literature. Towards a better understanding and evaluation of these proposals, this paper discusses how object concepts are articulated (applied, customized, integrated, expanded, and so on) 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 in which the object paradigm is used in concurrent and distributed contexts.

1.1 A Coarse Classification in Three Approaches

By analyzing current experience 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.

The *library* approach (Section 2) applies object-oriented concepts, as they are, to structure concurrent and distributed systems through class libraries. Various components, such as processes, synchronization means, files, and name servers, are represented by various object classes (services). 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 (Section 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 synchronization and replication with communication (see Section 3.6).

The *reflective* approach (Section 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 de-

scribed in terms of *metaprograms*. This makes possible (dynamic) system customization with minimal impact on the application programs.

1.2 Complementarity of the Approaches

Although these approaches may at first glance appear in conflict, in fact they are not. More precisely, research directed along these approaches has 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 a "bridge" between the two previous approaches as it helps to integrate transparently various computing protocol libraries within a programming language/system. Moreover, it helps in combining the two other approaches by making explicit the separation of and interface between their respective levels (i.e., the integrative approach for the end user and the library approach for developing and customizing 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 point out, for example, in Section 5.4, some languages and systems are built following more than one approach.

1.3 Concurrency and Distribution

Before presenting our classification in more detail, we first briefly clarify our use of 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, for example, 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 more 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). The situation with client/server systems is similar: 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 do we see a concurrent system operating in a distributed fashion. Second, distribution intrinsically implies independent failures; that is, 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 it is.

1.4 Previous Work

The reader is assumed to be familiar with traditional concurrency and distribution concepts, such as described in Andrews [1991]. It should also be kept in mind that object-based concurrent and distributed programming are well-established disciplines that are supported by many languages. On the one hand, Ada [1983] is a well-known example of a concurrent object-based lan-

guage and SR [Andrews et al. 1993], although less known, features versatile and powerful concurrency constructs. Argus [Liskov and Sheifler 1983] and Emerald [Black et al. 1987] have become known for their distribution support.

Combining concurrency and distribution with object orientation proper, that is, 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; 1995]. An early book featuring different articles on concurrent object-oriented programming is by Yonezawa and Tokoro [1987]; a more recent one is by Agha et al. [1993]. Furthermore, several workshops have been devoted to object-based concurrency and distribution: see Agha [1989]; Agha et al. [1991]; Briot et al. [1995]; Guerraoui et al. [1994]; and Tokoro et al. [1992].

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 into various components with clear interfaces. This improves structuring of concurrent and distributed systems, as opposed to UNIX-style 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 the microkernel. The architecture of the (generic) distributed operating system is organized along abstract notions of class components, which may then be specialized 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 and for distributed programming in two different well-known object-oriented languages: (1) the Smalltalk-80 programming language and environment, where a basic and simple object concept is uniformly applied to model and structure the whole system through class libraries, including concurrency and distribution aspects; and (2) C++, whose widespread use has resulted in a proliferation of concurrency libraries [Wilson and Liu 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 credo 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 (control structures, data structures, and so on), internal resources (messages, processes, compiler, and so on), and a sophisticated programming environment with integrated tools (browser, inspector, debugger, and so on).

Actually, even basic control structures, such as loop and conditional, are not primitive language constructs, but just standard methods of standard classes that make use of the generic

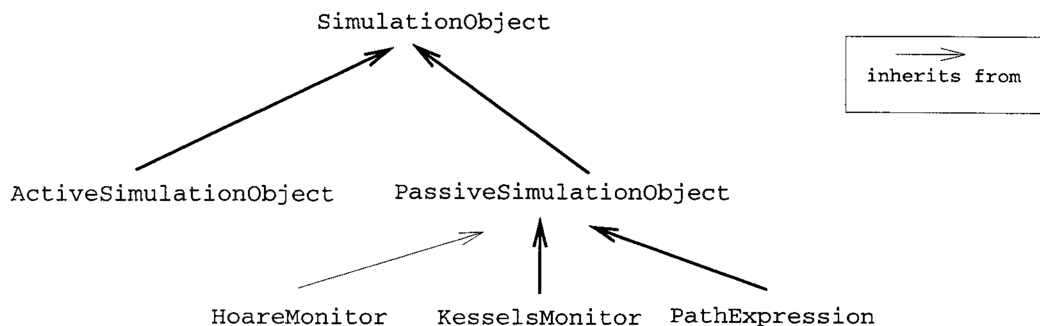

Figure 1. Hierarchy of concurrency classes in Simtalk.

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 at his wish.

2.2.1 Libraries for Concurrent Programming

In Smalltalk, blocks are also the basis for multithreaded concurrency through processes. The standard class `Process` describes their representation and its associated methods implement process management (suspend, resume, adjust priority, and so on). The behavior of the process scheduler is itself described by the class `ProcessorScheduler`. The basic synchronization primitive is the semaphore, represented by the 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 organized in a class hierarchy. Thus, they are much more understandable and extensible than if they were just simple primitives of a programming language. Furthermore, it is relatively easy to build up on the basic standard library of concurrency classes to construct more sophisticated abstractions, as for example in the Simtalk [Bézivin 1989] or Actalk [Briot 1989] platforms. The Simtalk platform implements and classifies various synchronization 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 parametrize 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 standard Smalltalk Binary Object Streaming Service (BOSS) library provides 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; 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 (point-to-point,

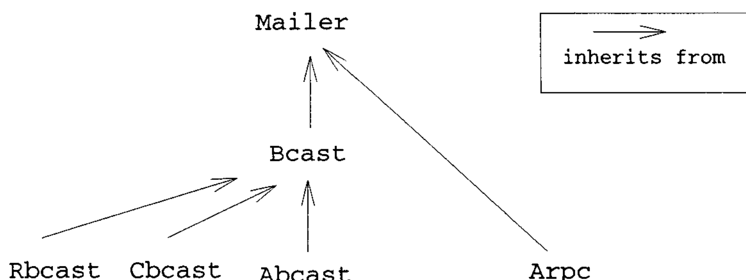

Figure 2. Communication in GARF.

multicast, atomic multicast . . .) and object models (monitor, persistent, 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 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 `RMPObject`, respectively, for 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. She also likes to exploit the low-level mechanisms and services offered by the underlying execution platform. As the library approach allows any functionality of a given platform to be wrapped in C++ functions or classes, it is not surprising that a wide variety of concurrency and distribution mechanisms are 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, heavyweight or lightweight, and for their corresponding synchronization mechanisms. Many concurrent programs are conveniently implemented using a threading system (e.g., network servers, interactive programs, parallel programs). We look first into object-oriented threading libraries.

Representation of threads. Although defining classes for synchronization 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,

```
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.

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 initialization 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 its own lifetime (as an object).

An example of the first approach is the *coroutine* part of Sun's C++ library [Sun 1995]. The library offers a class `task` (i.e., this plays the role of the class `Thread` in the preceding classification). A `task` object is implemented as a coroutine with nonpreemptive scheduling. There is also a class `Interrupt_handler` that allows catching UNIX software interrupts (signals). Typical operations on tasks are `result()` (wait for termination), `rdstate()` (get state), and the like. Synchronization 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

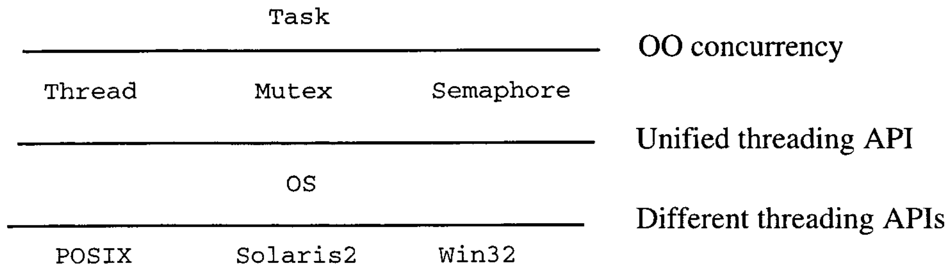

Figure 4. Architecture of the ACE concurrency library.

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 synchronization, PRESTO features atomic integers and lock, monitor and condition classes.

An alternative is 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 also offers a few classes for concurrent programming. Concurrency is implemented 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 synchronization. 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 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 synchronization. A class template `Guard` is parameterized with a lock class (e.g., `Mutex`). A guard object acquires and releases a lock upon initialization and finalization, respectively, similarly to a PRESTO monitor object; thus, declaring a guard in a block turns 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` (see Figure 5), but this is just a package of static functions such as `spawn`, `join`, `yield`, and the like, abstracting from the idiosyncrasies of the threading functions of POSIX, Solaris, and Win32. Another class, `Thread_Manager`, serves 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, and so on. But there is no class resembling Smalltalk's `Process` or the `task` from Figure 3.

```
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.

A relatively high-level concept in ACE is the `Task` class. This class must not be confused with Sun's `task` class mentioned previously. `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 the 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 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. Although 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, for example, MPI [Skjellum et al. 1996]. Libraries of this kind can be seen as the "parallel" equivalent to threading libraries as described previously.

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

Choices. Choices [Campbell et al. 1993] is a *generic* operating system, of which the objective is not only to be easily ported onto various machines, but also to be able to adjust various characteristics of both hardware, resources, and application interfaces such as: file format, communication network, and memory model (shared or distributed). An object-oriented methodology is presented together with the system, for the design both of distributed applications and of new extensions to the Choices kernel.

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

The experience of the Choices projects shows that a distributed operating system, developed with an object-oriented methodology and programming language (C++ in this case), helps in achieving better genericity and extensibility.

Peace. Similar in spirit to Choices, the Peace parallel operating system [Schröder-Preikschat 1994] has as its target 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 simply by 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 of 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 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 synchronization for concurrent programming. Such a basic abstraction may be used as a foundation to build various higher-level synchronization mechanisms (e.g., the Guard class of ACE). Classification and specialization mechanisms, as offered by object-oriented programming, are then appropriate for organizing such a library/hierarchy of abstractions, as for instance in the Simtalk platform (Section 2.2). Peace is a typical example of 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 synchronization patterns between processes. The abstract specifications and the relative independence of 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 [Brandt and Lehrmann-Madsen 1994] in the Beta programming language [Lehrmann-Madsen et al. 1993]. For instance, class NameServer represents a name server that 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/centralized program without changing the program logic, that is, through additions rather than changes [Brandt and Lehrmann-Madsen 1994, p. 199].

A fundamental study of abstractions for distributed programming has been proposed by Black [1991], where decomposing the concept of transaction into a set of abstractions is suggested. 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 in 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] The BAST project [Garbinato and Guerraoui 1997] aims at defining reliable distributed protocols from a set of minimal abstractions. One of these abstractions, the consensus, plays the role of the semaphore in a distributed context.

3. AN INTEGRATIVE APPROACH

3.1 Unification Needs

The number of issues and concepts required is one of the major difficulties of

[1] The former application requires strict serialization of transactions through a locking mechanism, whereas the latter does not.

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 in structuring concurrency and distribution concepts and mechanisms, but keeps them disjoint from the objects structuring the application programs. In other words, the programmer still faces at least two different major 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 to the standard programming style. For instance, a library implementing asynchronous and remote communication in Eiffel will force the programmer to do some amount of explicit message manipulation (see Karaorman and Bruno [1993, pp. 109–111]), 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, for example, Eiffel// [Caromel 1990], or to define a brand-new language containing that construct.

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

3.2 Dimensions of Integration

There are various possible levels of integration between object-oriented programming concepts and concurrency and distribution concepts. Here we distinguish three main levels. Note that they are relatively independent of each other. Thus, as we show, a given language or system may follow one dimension of integration but not another.

(1) A first level 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 (as noted in Meyer [1993]). 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 level of integration associates synchronization with object activation, leading to the notion of a synchronized object. Message passing is then considered an implicit synchronization between the sender and the receiver. Furthermore, one often associates mechanisms for controlling the activation of invocations at the level of an object, for example, by attaching a guard to each method. Note that the concept of an active object already implies some form of synchronized object, as the existence of a (single) activity private to the object actually enforces the serialization of invocations. However, some languages or systems, such as Guide [Balter et al. 1994] or Arjuna [Parrington and Shrivastava 1988], associate synchronization with objects although they distinguish the notions of object and autonomous activity. Another more recent example is Java [Lea 1997], where a new private lock is implicitly associated with each newly created object.

(3) A third level of integration considers the object as the unit of distribution, leading to the notion of a distributed object. Objects are seen as entities that 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. Emerald [Black et al. 1987] is an example of a distributed programming language based on the notion of distributed object. One can also further integrate message passing with the transaction concept, so as to support interobject synchronization 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, that is, its own private activity. This approach, simple and natural, is quite influential [Yonezawa and Tokoro 1987], following the course of actor languages [Lieberman 1987; Agha 1986].

3.3.1 *Levels of Object Concurrency.* The independence of object activities provides what is usually called interobject concurrency. Some languages (e.g., 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 *intraobject* concurrency.

More generally, one may consider different levels of object concurrency. Our classification extends that of Wegner [1990]. An active object may be:

—*Serial* or *atomic*. Only one message is computed at a time. Examples of languages offering serial active objects are POOL [America 1987] and Eiffel// [Caromel 1990].

—*Quasiconcurrent.* 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 are ABCL/1 [Yonezawa et al. 1986] and ConcurrentSmalltalk [Yokote and Tokoro 1987].

—*Concurrent.* There is true intraobject concurrency but some degree of control (i.e., restrictions) applies, as specified by the programmer. Example languages are actor languages such as ACT++ [Kafura and Lee 1990] and also CEiffel [Löhr 1993].

—*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, where they are called *unserialized actors.*

Setting the possible level (or levels) of object-internal concurrency for a programming language is a design decision, and an important one. Some researchers have argued that intraobject concurrency should be banished altogether because reasoning about programs that contain only serial objects 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 intraobject 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.

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

3.3.2 *Reactivity Versus Autonomy.* One of the important characteristics of object-oriented programming is the *reactivity principle* (as stated by Kay in the late ’60s [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 of whether reactivity will be preserved or shadowed by the autonomous behavior of the process. We may distinguish two families of active objects:

—*Reactive active object.* It adheres to the reactivity principle, and thus can be activated only through message passing. Example languages are actors, as in ACT++ and also CEiffel.
—*Autonomous active object.* It may compute before being sent a message. Example languages are POOL and Eiffel//.

Although the models are opposite, please note that they can 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 Section 3.3.4, where this is actually the default case.)

3.3.3 *Implicit Versus Explicit Acceptance of Messages.* Another issue 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 after it is received (the actual processing may be delayed after receipt of the message because of synchronization requirements). Explicit acceptance means that the object explicitly states that it is willing to accept a certain pattern of message. This is analogous to the task entry statement in Ada.

In looking at the relation between the reactivity versus autonomy issue and the implicit versus explicit acceptance issue, one's 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.

—ABCL/1 is a programming language based on the concept of reactive active object that offers explicit message acceptance from within a method.
—CEiffel supports autonomous objects. But methods are activated implicitly, independently of any explicit message acceptance (see Section 3.3.5).
—The synchronization scheme, abstract states, (described in Section 3.4.3) combines implicit acceptance with some explicit matching of a message.

3.3.4 *The Concept of a Body.* Most languages following the model of an autonomous active object are based on the concept of a *body:* some distinguished centralized operation that explicitly describes the types and sequence of requests the object will accept during its activity. This concept is actually a direct offspring of the Simula-67 [Birtwistle et al. 1973] concept of body, which included support for coroutines. This initial potential of objects for concurrency was, however, abandoned, for both technological and cultural reasons, by most 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[2] and Eiffel// [Caromel 1993] are typical representatives of the body concept. A POOL class for active `Queue`

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

```
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.

objects is shown in Figure 6. The declarative part for local data is omitted. Operations such as `METHOD enq` are declared just as 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 as 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, there may be some alternatives. In CEiffel, operations can be specified as autonomous using the autonomy annotation `-->--`. Note that a CEiffel annotation always starts with the characters `--`, just like an Eiffel comment,

```
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. Modeling autonomous moving objects in CEiffel.

but is identified as an annotation by the next character. Being comments, annotations are not interpreted by a standard Eiffel compiler, but only by a specific CEiffel precompiler (and associated runtime) [Löhr 1993].

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 distinguish between explicit and implicit invocations. Note that the degree of intraobject concurrency is controlled in CEiffel by other kind of annotations, namely compatibility annotations (see Section 3.4.3).

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

3.4 Synchronized Objects

The presence of concurrent activities requires some degree of synchronization, that is, constraints, in order to ensure correct program execution. Synchronization may be associated with objects

```
server: ActiveServer;

result: R;

.....

result := server.service(args);

.....         -- client continues immediately

result.op     -- synchronisation is implicit
```

Figure 8. Synchronization by need in Eiffel//.

and with their communication means (i.e., message passing) through various (sub-)levels of identification.

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

In the case of active objects, the sender and the receiver own independent activities. It is therefore useful to introduce some *asynchronous* type of transmission by which the sender resumes its execution as soon it has sent the message, that is, 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 with an active object a mail queue that will buffer incoming messages (usually in the ordering of their arrival) before the active object is ready to compute them.

Finally, some languages (e.g., ABCL/1 and ACT++) introduce some mixed kind of transmission that immediately returns an eager 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 caller really needs the result—that is, is going to operate on it—is synchronization with the service provider required. Integration of futures into the invocation mechanism has the effect that the strict synchronization inherent in synchronous invocation is replaced with synchronization by need, also sometimes called lazy synchronization. The concept of future originated in the actor languages. It was included as a specific type of message transmission in early object-oriented concurrent languages, notably ABCL/1,[3] and was later integrated into some extensions of existing languages, like Eiffel// [Caromel 1990] where it is known as wait-by-necessity.

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

3.4.2 *Synchronization at the Object(s) Level.* The identification of synchroni-

[3] ABCL/1 actually offers three types of message transmission: synchronous (called now type), asynchronous (called past type), and with eager-reply (called future type).

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

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

Intraobject Synchronization. In *intraobject* concurrency (i.e., an object simultaneously computing several requests), it is necessary to include some concurrency control in order to ensure the consistency of the object state. Usually, the control is expressed in terms of exclusions between operations.[4] The typical example is the readers and writers problem, where several readers are free to access simultaneously a shared book but the presence of one writer excludes all others (writers and readers). Note that intraobject synchronization is the equivalent at the object level of what is called (mutual) exclusion synchronization at the data level.

Behavioral Synchronization. It is possible that an object may temporarily be unable to process a certain kind of request that 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 synchronization of services between objects fully transparent. Behavioral synchronization is the equivalent at the object level of what is called condition synchronization at the data level. (Andrews [1991] is an excellent book on synchronization principles and terminology.)

Interobject Synchronization. Finally, it may be necessary to ensure some consistency, not just individual but also global (coordination) between mutually interacting objects. Consider a money transfer between two bank accounts. The issue is ensuring the invisibility of possible transient and inconsistent global states while the transfer takes place. Intraobject or behavioral synchronization are not sufficient, and a notion such as an atomic transaction [Bernstein et al. 1987] is needed to coordinate the different invocations.

3.4.3 *Synchronization Schemes.* Various synchronization schemes have been proposed to address these different 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 may make a general distinction with regard to whether synchronization specifications are centralized.

—*Centralized schemes,* such as *path expressions,* specify in an abstract and centralized way the synchronization of the object. As they are centralized, they tend to be naturally associated and then integrated with the class.
—*Decentralized schemes,* such as *guards,* specify at the program-area level the synchronization of the object. As they are decentralized, they tend to be naturally associated and then integrated with a method.

There has been considerable debate about the pros and cons of various schemes in these two general categories. Important issues are: expressivity, reusability, provability, and efficiency. Several variations are described in the following, always with a focus on how

[4] Note that the case of a mutual exclusion between all methods subsumes the case of a serialized object (as defined in Section 3.3.1).

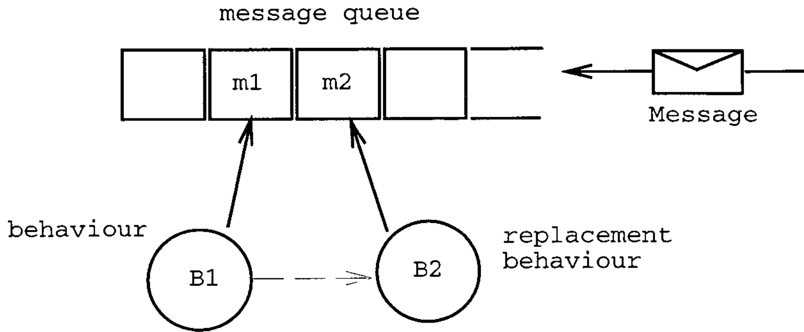

Figure 9. Behavior replacement of an actor.

they can be integrated with the object concepts (for a more detailed and exhaustive presentation, please see Andrews [1991]). Note that most activity, synchronization, and communication models described in these sections have been implemented as various component libraries in a common framework for object-oriented concurrent programming called Actalk [Briot 1996]. This framework provides a relatively neutral foundation and platform to study, compare, and experiment with various models of activity (reactivity, body, etc.), communication (synchronous, asynchronous, etc.) and synchronization.

Path Expressions. A first example of a centralized scheme is the concept of *path expressions,* which specifies in a compact notation the possible interleaving 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 centralized scheme is the concept of body, described Section 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 centralized handling in the body plus its imperative nature, are the source of several problems when specializing object behaviors. In most cases the body needs to be rewritten from scratch. The general problem of reusing synchronization specifications is 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 last concept is both simple and very expressive. When created, an actor is composed of an address—with 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 computes the first incoming message[5] and specifies the *replacement behavior,* that is, the behavior that will compute the next message (see Figure 9). As one may imagine, once triggered by an incoming message, the replacement behavior will specify in turn its own replacement behavior, and so on.

Note that as soon as the replacement behavior is specified, the computation of the next message may start. This implies intraobject concurrency. Conversely, as long as the replacement behavior is not specified, the computation of the next message will not proceed. This implies synchronization, to be

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

more precise, intraobject synchronization.

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 synchronization described previously (i.e., to delay the acceptance of a request until a service is available; see Section 3.4.2[6]), and if we assume that the active object is serial, we obtain the concept of abstract states.

The idea is the following: an object conforms to some abstract state representing a set of enabled methods (see, e.g., Matsuoka and Yonezawa [1993] for a more detailed description). In the example of the bounded buffer, three abstract states are needed: `empty`, `full`, and `partial`. 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. After completing the processing of an invocation, the next abstract state is computed to possibly update the state and services availability of the object.

The corresponding program in ACT++, which is based on this idea, is shown in Figure 10.

Guards. The notion of *guard* is a major example of a decentralized synchronization scheme. A guard is basically a Boolean activation condition that 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 synchronization statements in the implementation of the object's operations. Activities are blocked or awakened implicitly. The price that must 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 Figure 11, lower part).

Synchronization counters are counters recording the invocation status for each method, that is, the number of received, started, and completed invocations. Associated with guards, they provide very fine-grained control of intraobject synchronization. Referring to counters has the advantage of representation-independence.

Note that, although integration of synchronization schemes with object models is usually straightforward, this integration has an impact on the reuse of synchronization specifications (see Section 3.6.1).

Locks. The concept of a lock is one of the basic synchronization abstractions. It is very natural to associate a lock with each object (or two locks, in order to distinguish between readers and writers methods) in order to make it a synchronized object. This is the approach followed by the Java programming language. Like Guide, Java is partially integrated in that it follows a model of synchronized objects, but not a model of active objects (object and thread are kept separate). A private lock is implicitly associated with each Java object at its creation. Although this basic synchronization abstraction is centralized in the object, the interface for the programmer is decentralized. Qualifying a method with the `synchronized` keyword indicates that this method is mutually exclusive with other qualified methods of the same class (or superclass). Actually, the `synchronized` keyword can also be used for establishing arbitrary critical regions. Condition synchronization is handled using events, another indication that Java favors a low degree of integration.

A language that allows specifying reader/writer exclusion is Distributed

[6] 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 that 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++.

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 generalized in another Eiffel extension, CEiffel [Löhr 1993]: using annotations to the operations, a binary, symmetric compatibility relation among the opera-

```
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`.

tions of an object can be specified. If operation `op1` is declared compatible with operation `op2`, both can be executed in an overlapping fashion. Incompatible operations are mutually exclusive. This approach can be traced back to Andrews [1991], where a centralized `parallel` clause is used to specify compatibility in a precursor of the SR programming language. Note that declaring compatibilities is safer than declaring exclusion requirements.

(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

```
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.

consider an object as the unit of distribution and possible replication. Furthermore, self-containedness of objects (data plus procedures, plus possible internal activity) eases the issue of moving and migrating them around. Also, note that message passing not only ensures the separation between services offered by an object and its internal representation but also provides the independence of its physical location. Thus, message passing may subsume both local and remote invocation (whether sender and receiver are on the same or distinct processors is transparent to the programmer) as well as possible 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.[7]

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 that may be placed on several processors, without any addition to the Eiffel language.

The essentials of ÉPÉE are as follows.

(1) A data aggregate is an Eiffel [Meyer 1991] 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 nondistributed class, say `Matrix`, and the class `DISTAGG`.
(3) The original operations of `Matrix` must 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 interfragment data exchange on remote read operations and provides various support functions such as fragment-specific index mapping.
(4) There is no explicit process creation or 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

---

[7] 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 object (see Section 3.3). Indeed, objects represent duality (and unification) between data and procedures (potential activation).

```
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.

fragments to start operating in parallel.

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

Charm++. The Charm++ language supports both the MIMD and SPMD style. 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 interfragment 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 with an invocation. If an object is located on a processor that is inaccessible because of a network or processor fault, an exception is raised, for example, 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 canceled. The Karos distributed programming language [Guerraoui et al. 1992] extends the Argus approach by allowing the association of nested transactions to 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. 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 ("called-by move"), or temporary ("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, and truly object-oriented, system named Dowl [Achauer 1993], a distributed extension of the Trellis/Owl language [Moss et al. 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, that is, to ensure that all replicas hold the same values. In the Electra [Maffeis 1995] distributed system, the concept of remote invocation has been extended in the following fashion. Invoking an object leads to the invocation of all its replicas while ensuring that concurrent invocations are ordered along the same (total) order for all replicas. Black and Immel [1993] also introduced a general mechanism for group invocation well suited for replicated objects.

3.6 Limitations of the Integrative Approach

The integrative approach attempts to unify object mechanisms with concurrency and distribution mechanisms. This integration leads, however, to some conflicts that we discuss in the following.

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 specialize synchronization specifications associated with a class of objects. Unfortunately, experience shows that: (1) synchronization is difficult to specify and moreover to reuse because of the high interdependency among the synchronization conditions for different methods, (2) various uses of inheritance (to inherit variables, methods, and synchronizations) may conflict with each other, as noted in McHale [1994] and Baquero et al. [1995]. In some cases, defining a new subclass, even only with one additional method, may force the redefinition of all synchronization specifications. This limitation has been named the *inheritance anomaly phenomenon* [Matsuoka and Yonezawa 1993].

Specifications along centralized schemes (see Section 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 Figure 6) named `ExtendedQueue`. A new operation `delete` for deleting the last element has been added. Note that there is no way of reusing the body of the superclass; a complete redefinition is required. Recognizing 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.

Decentralized schemes, being modular by essence, are better suited for selective specialization. However, this

```
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.

fine-grained decomposition, down at the level of each method, is only partially successful. This is because synchronization specifications, even if decomposed for each method, still remain more or less interdependent. As, for instance, in the case of intraobject synchronization with synchronization counters, adding a new write method in a subclass may force redefinition of other methods' guards in order to take into account the new mutual-exclusion constraint.

The fact that intraobject concurrency (exclusion synchronization) is implementation-dependent and therefore conceptually different from behavioral synchronization (condition synchronization) is recognized by some, though not all, languages. Guide employs counters not only for condition synchronization, as mentioned previously, but also for exclusion synchronization. Unfortunately, this approach makes specifications more difficult to reuse. Figure 15 shows the definition in Guide of the same subclass `ClearableQueue` (see the original definition of `Queue` in Guide in Figure 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.

The various cases of inheritance anomaly have been carefully studied and classified in Matsuoka and Yonezawa [1993].

Among the recent directions proposed for minimizing the problem, we may cite:

(1) specifying and specializing independently behavioral synchronization and intraobject synchronization [Thomas 1992] as well as autonomy/asynchrony [Löhr 1993];
(2) specifying disabled methods rather than enabled methods, as usually disabled methods remain disabled in subclasses [Frølund 1992];
(3) allowing the programmer to select among several schemes [Matsuoka and Yonezawa 1993];
(4) offering a framework to help in designing, customizing, and combining between various synchronization schemes [Briot 1996]; and
(5) instantiating abstract specifications as an alternative to inheritance for reusing synchronization specifications [McHale 1994]. Another example of a high-level approach is the coordination patterns proposed by Frölund [1996], with a specific focus on interobjects synchronization. These abstractions are specified in-

```
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.

dependently of how an implementation can be (automatically) derived in order to help in reusing them.

3.6.2 *Compatibility of Transaction Protocols.* It is tempting to integrate transaction concurrency control protocols into objects so that one could locally define, for a given object, the optimal concurrency control or recovery protocol. For instance, commutativity of operations makes possible the interleaving (without blocking) of transactions on a given object. Unfortunately, the gain in modularity and specialization may lead to incompatibility problems [Weihl 1989]. Broadly speaking, if objects use different transaction serialization protocols (i.e., serialize the transactions along different orders), global executions of transactions may become inconsistent, that is, nonserializable. A proposed approach to that problem is to define 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 that have been designed for fault-tolerant distributed computing (see Section 3.5.4) consider a standard client/server model. The straightforward transposition of such protocols to the object model leads to unexpected duplication of invocations. Indeed, an object usually acts conversely as a client and as a server. Thus an object that 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 ineffi-

ciency, and in the worst case to inconsistencies (by invoking the same operation several times). A solution proposed in Mazouni et al. [1995] is based on *prefiltering* and *postfiltering*. Prefiltering consists of coordinating processing by the replicas (when considered as a client) in order to generate a single invocation. Postfiltering is the dual operation for the replicas (when considered as servers) in order to discard redundant invocations.

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

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

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

A radical approach replaces the inheritance mechanism between classes by the concept/mechanism of *delegation* between objects. This mechanism was actually introduced in the Actor concurrent programming language Act 1 [Lieberman 1987]. Intuitively, an object that may not understand a message will then delegate it (i.e., forward it) to another object called its *proxy*. (Note that, in order to handle recursion properly, the delegated message will include the initial receiver.) The proxy will process the message in place of the initial receiver, or it can also delegate it itself further to its own designated proxy. This alternative to inheritance is very appealing as it relies only on message passing and hence fits well with a distributed implementation. Meanwhile, the delegation mechanism needs some nontrivial synchronization mechanism to ensure the proper handling (ordering) of recursive messages, prior to other incoming messages. Thus, it may not offer a general and complete alternative solution [Briot and Yonezawa 1987].

4. THE REFLECTIVE APPROACH

4.1 Combining Flexibility and Transparency

As discussed earlier, the library approach (library-based approach) helps in structuring concurrent and distributed programming concepts and mechanisms, thanks to encapsulation, genericity, class, and inheritance concepts. The integrative approach minimizes the number 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 with an integrative approach. In other words, libraries help structure and simulate various solutions and thus usually bring good flexibility, whereas brand-new languages may freeze their computation and communication models too early. 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 programmer. The end user programs applications with an integrative (simple and unified) approach in mind. The system programmer, or the more expert user, builds or customizes the system, through the design of libraries of protocol components, along a library approach.

Therefore—and as opposed to what one may think at first glance—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? To be more precise: how do we interface them? It turns out that a general methodology for adapting the behavior of computing systems called reflection offers such a glue.

4.2 Reflection

Reflection is a general methodology for describing, controling, and adapting 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 made concrete in one (or more) program(s) that represent the default computational behavior (interpreter, compiler, execution monitor, and so on). Such a description/control program is called a metaprogram. Specializing such programs enables us to customize the execution of the application program, by possibly changing data representation, execution strategies, mechanisms, and protocols. Note that the same language is used both for writing application programs and for metaprograms controlling their execution. However, the complete separation between the application program and the corresponding metaprograms is strictly enforced.

Reflection helps in decorrelating libraries specifying implementation and execution models (execution strategies, concurrency control, object distribution) from the application programs. This increases modularity, readability, and reusability of programs. Reflection also provides a methodology for opening up and making adaptable, through a meta-interface,[8] implementation decisions and resource management that are often hard-wired and fixed or delegated by the programming language to the underlying operating system.

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

4.3 Reflection and Objects

Reflection fits especially well with object concepts, which enforce good encapsulation of levels and modularity of effects. It is therefore natural to organize the control of the behavior of an object-oriented computational system (its meta-interface) through a set of objects. This organization is named a meta-object protocol (MOP) [Kiczales et al. 1991], and its components are called *meta-objects* [Maes 1987], as metaprograms are represented by objects. They may represent various characteristics of the execution context such as: representation, implementation, execution, communication, and location. Specializing meta-objects may extend and modify, locally, the execution context of some specific objects of the application program.

Reflection may also help in expressing

[8] 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 termed the concept of open implementation by Kiczales [1994].

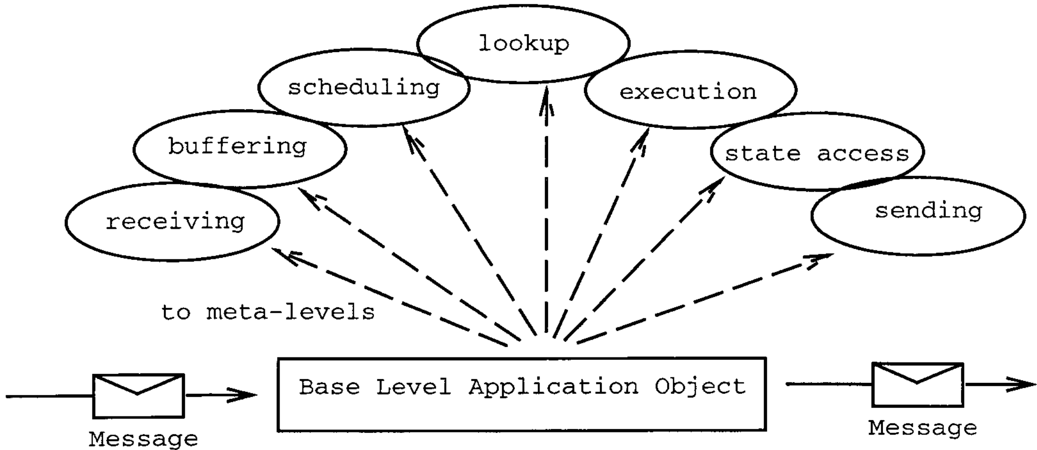

Figure 16. Metacomponents in CodA.

and controlling resource management, not only at the level of an individual object but also on a broader level such as scheduler, processor, name space, object group, and the like, such resources being also represented by meta-objects. This helps with the 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 optimized for a specific kind of application or an average case).

4.4 Examples of Meta-Object Protocols (MOPs)

Smalltalk. Depending on the actual goals and the balance expected among flexibility, generality, simplicity, and efficiency, design decisions will dictate the amount and scope of the mechanisms to be opened up to the metalevel. Therefore, some mechanisms may be represented as reflective methods but belong to standard object classes, that is, without explicit and complete meta-objects.

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

CodA. The CodA architecture [McAffer 1995] is a representative example of a general object-based reflective architecture (i.e., a MOP) based on metacomponents. (Note that metacomponents are indeed meta-objects. In the following, we prefer the term *metacomponent* to emphasize the pluggability aspects of a reflective architecture (MOP) such as CodA. Also, for simplificity, we often use the term *component* in place of *metacomponent.*) CodA considers by default seven metacomponents associated with each object (see Figure 16), corresponding to: message sending, receiving, buffering, selection, method lookup, execution, and state accessing. An object with default metacomponents 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 (specialized) metacomponents allows selectively changing a specific aspect of the representation or execution model for a single object. A standard interface between metacomponents helps in composing metacomponents from different origins.

Actalk and GARF. Some other reflective architectures may be more specialized and may offer a more reduced (and abstract) set of metacomponents. Examples are the Actalk and GARF platforms, where fewer metacomponents may be in practice sufficient to express a large variety of schemes and application problems.

The Actalk platform [Briot 1989; 1996] helps in experimenting with various synchronization and communication models for a given program by changing and specializing various models/components of: (1) *activity* (implicit or explicit acceptance of requests, intraobject concurrency, etc.) and *synchronization* (abstract behaviors, guards, etc.), (2) *communication* (synchronous, asynchronous, etc.), and (3) *invocation* (time stamp, priority, etc.). 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 (persistence, replication, etc.) and (2) communication (multicast, atomic, etc.).

4.5 Examples of Applications

To illustrate how reflection may help in mapping various computation models and protocols onto user programs, we quickly survey some examples of experiments with a specific reflective architecture. (We chose CodA; see McAffer [1995] for a more detailed description of its architecture and libraries of components.)

Note that, in the CodA system, as well as almost all other reflective systems described, the basic programming model is integrative, whereas reflection makes possible the customization of concurrency and distribution aspects and protocols by specializing libraries of metacomponents.

4.5.1 *Concurrency Models.* In order to introduce concurrency for a given object (by making it into an active object, in an integrated approach), two metacomponents are specialized: the specialized message-buffering component[9] is a queue to buffer incoming messages, and the specialized execution component associates an independent activity (thread) with the object. This thread processes an endless loop for selecting and performing the next message from the buffering component.

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

4.5.3 *Migration and Replication Models.* Migration is introduced by a new metacomponent describing the form and the policies (i.e., when it should occur) for migration. Replication is managed by adding two new dual metacomponents: the first is in charge of controlling access to the state of the original object, and the other controls access to

[9] The default buffering component actually passes on incoming messages directly to the execution component.

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 specialized through the marshaling descriptors supplied by the corresponding component. One may also specialize such aspects as which parts of the object should be replicated, and various management policies for enforcing consistency between the original object and its replicas.

4.6 Other Examples of Reflective Architectures

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

4.6.1 *Dynamic Installation and Composition of Protocols.* The general MAUD methodology [Agha et al. 1993] focuses on fault-tolerance protocols, such as server replication and check point. Its strength is in offering a framework for dynamic installation and composition of specialized metacomponents. The dynamic installation of metacomponents makes possible the installation of a given protocol only when needed and without stopping program execution. The possibility of associating metacomponents not only to objects but also to other metacomponents (which are first-class objects) makes possible the layered composition of protocols. A higher-level layer for specification of dependability protocols (called DIL [Sturman and Agha 1994]) has been designed that makes use of the underlying MAUD reflective architecture.

4.6.2 *Control of Migration.* The autonomy and self-containedness of objects, further reinforced in the case of active objects, makes them easier to migrate as a single piece. Nevertheless, the decision to migrate an object is an important issue that often remains the programmer's responsibility (e.g., in Emerald [Black et al. 1987]). It may be interesting to semi-automate such decisions using various considerations such as processor load and ratio of remote communications. Reflection helps in integrating such statistical data (residing for physical and shared resources) and in using them by various migration algorithms described at the metalevel [Okamura and Ishikawa 1994].

4.6.3 *Customizing System Policies.* The Apertos distributed operating system [Yokote 1992] is a significant and innovative example of a distributed operating system completely designed along an object-based reflective architecture (MOP). Supplementary to the modularity and the genericity of the architecture gained by using a library (object-oriented) approach (as for Choices, already discussed in Section 2.3.2), reflection brings the (possibly dynamic) customization of the system towards application requirements; for instance, one may easily specialize the scheduling policy in order to support various kinds of schedulers, such as a real-time scheduler. Another gain is in the size of the microkernel obtained, which is particularly small, as it is reduced to supporting the basic reflective operations and the basic resource abstractions. This helps in both understanding and porting the system.

4.6.4 *Reflective Extension of an Existing Commercial System.* A reflective methodology has recently been used in order to incorporate extended (i.e., relaxing some of the standard (ACID) transaction properties) 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 1995]. These reflective primitives are then used to implement

various extended transaction models, such as: split/join, cooperative groups, and the like.

4.7 Related Models

We finally mention two models for customizing computational behaviors that are closely related to reflection.

4.7.1 *The Composition-Filters Model.* The SINA language is based on the notion of a *filter*, a way to specify arbitrary manipulation and actions for messages sent to (or from) an object [Aksit et al. 1994]. 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.

4.7.2 *Generic Run-Time as a Dual Approach.* The frontier between programming languages and operating systems is getting thinner. Reflective programming languages have some high-level representation of the underlying execution model. Conversely, and dual to reflection, several distributed operating systems provide a generic run-time layer such as the COOL layer in the Chorus operating system [Lea et al. 1993]. These generic run-time layers are designed to be used by various programming languages, thanks to some upcalls that 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 objectives are very important because concurrent and distributed computing systems are complex systems that ultimately use very low-level mechanisms, for example, network communication. Furthermore, such systems are often developed by teams of programmers, and in such a context, having separate modules with well-defined interfaces is of primary importance. The difficulty in maintaining and extending UNIX-like systems comes mainly from their low modularity and insufficient level of abstraction.

Although progress is being made in 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 abstraction became a standard primitive for synchronization leads us to think, however, that other abstractions for concurrent and distributed programming can also be identified and adopted.

5.2 The Integrative Approach

The integrative approach is very appealing in its merging of concepts from object-oriented programming and those from concurrent and distributed programming. It thus provides a minimal number of concepts and a single conceptual framework to the programmer. Nevertheless, this approach unfortunately has limitations in some aspects of the integration (e.g., inheritance anomaly and 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 also lead to inefficiencies. For instance, stating that every object is active and/or every message transmission is a transaction may be inappropriate for some applications not necessarily requiring such protocols, and their associated computational load. A last

important limitation is the legacy problem, that is, the possible difficulty of reusing standard sequential programs. A straightforward way of handling this is the encapsulation of sequential programs into active objects. However, cohabitation of active objects and standard ones (i.e., passive objects), is nonhomogeneous and requires specific methodological rules for distinction between active objects and passive objects [Caromel 1993].

5.3 The Reflective Approach

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

Many reflective architectures are currently being proposed and evaluated. It is too early yet to 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 large effort in designing reflective architectures (MOPs) for the C++ programming language. The goal is to offer some generic framework in order to express various models and protocols of parallel and distributed programming. Two significant examples of such efforts are OpenC++ [Chiba 1995] and C++// [Caromel et al. 1996].

Meanwhile, we still need more experience in the practical use of reflection in order to find good tradeoffs among the flexibility required, the architecture complexity, and the resulting efficiency. One possible (and currently justified) complaint concerns the actual relative complexity of reflective architectures. Nevertheless, and independently 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, as a consequence of extra indirections and interpretations. Some alternative directions are: (1) to reduce the scope of reflection at compile-time as shown by the evolution of the initial reflective OpenC++ architecture into a compile-time reflective architecture, thus getting closer to metacompilers [Chiba 1995], or (2) to use program transformation techniques, notably partial evaluation (also called program specialization), to minimize metainterpretation overheads [Masuhara et al. 1995].

5.4 Integrating the Approaches

As remarked in Section 1.2, the library, integrative, and reflective approaches are not in conflict but are complementary. This complementarity extends to their relationship to language: the library approach does not change the underlying language but 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 article, some have been built following more than one approach. This is the case, for instance, of the ÉPÉE [Jézéquel 1993a] parallel system (see Section 3.5.1), which is based on the integration of object with distribution, and is also implemented with libraries. Other examples are Actalk [Briot 1989] and GARF [Garbinato et al. 1994] (see Section 2.2), which offer libraries of abstractions for concurrent and distributed programming that can be transparently applied to programs thanks to the reflective facilities of Smalltalk.

We believe that future developments in 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 customization of concurrency and distribution protocols. In that sense, CORBA implementations follow (to some extent) the reflective approach.

6. SUMMARY

In order to understand and evaluate various object-based concurrent and distributed developments, we have proposed a classification of the different ways in which the object paradigm is used in concurrent and distributed contexts. The three approaches we have identified convey different yet complementary research streams in the object-based concurrent and distributed system community.

The library approach (library-based approach) helps in 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 that of the concurrent and distribution architecture, is 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 minimizes the concepts to be mastered by the programmer and makes mechanisms more transparent by providing a unified concurrent and distributed high-level object model. However, this has 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 customizations and thus change the execution context (concurrent, distributed, fault-tolerant, real-time, adaptive, and so on) with minimal changes in the application programs.

The reflective approach contributes to blurring the distinction between programming language, operating system, and database, and at easing the development, adaptation, and optimization of a minimal dynamically extensible computing system. Nevertheless, we stress that this does not free us from the necessity of having a good basic design and finding a good set of foundational abstractions [Guerraoui et al. 1996].

ACKNOWLEDGMENT

The comments of the reviewers have greatly contributed to improving the quality of this article.

REFERENCES

ACHAUER, B. 1993. The Dowl distributed object-oriented language. *Commun. ACM 36,* 9.

ADA 1983. *The Programming Language Ada Reference Manual.* In Lecture Notes in Computer Science, vol. 155. Springer-Verlag, New York.

AGHA, G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence, MIT Press, Cambridge, MA.

AGHA, G. A., FRØLUND, S., PANWAR, R., AND STURMAN, D. 1993. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III (DCCA-3).* IFIP Transactions,

Elsevier-North Holland, Amsterdam, The Netherlands, 197–207.

AGHA, G. A., HEWITT, C., WEGNER, P., AND YONEZAWA, A., EDS. 1991. *Proceedings of the OOPSLA/ECOOP '90 Workshop on Object-Based Concurrent Programming, ACM OOPS Mess. 2,* 2.

AGHA, G. A., WEGNER, P., AND YONEZAWA, A., EDS. 1989. *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, ACM SIGPLAN Not. 24,* 4.

AGHA, G. A., WEGNER, P., AND YONEZAWA, A., EDS. 1993. *Research Directions in Concurrent Object-Oriented Programming,* MIT Press, Cambridge, MA.

AKSIT, M., WAKITA, K., BOSCH, J., BERGMANS, L., AND YONEZAWA, A. 1994. Abstracting object interactions using composition filters. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming,* Guerraoui et al., Eds. Lecture Notes in Computer Science, vol. 791. Springer-Verlag, New York, pp. 152–184.

AMERICA, P. H. M. 1987. Pool-T: A parallel object-oriented language. In *Object-Oriented Concurrent Programming,* A. Yonezawa and M. Tokoro, Eds. Computer Systems Series, MIT Press, Cambridge, MA.

AMERICA, P. H. M. 1988. Definition of Pool2, a parallel object-oriented language. ESPRIT project 415-A, Tech. Rep. 364, Philips Research Laboratories.

AMERICA, P. H. M. 1989. Issues in the design of a parallel object-oriented language. *Formal Aspects Comput. 1,* 366–411.

AMERICA, P. H. M. AND VAN DER LINDEN, F. 1990. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of OOPSLA/ECOOP '90, ACM SIGPLAN Not. 25,* 10.

ANDREWS, G. R. 1991. *Concurrent Programming—Principles and Practice.* Benjamin/Cummings, Redwood City, CA.

ANDREWS, G. R. AND OLSSON, R. A. 1993. *The SR Programming Language.* Benjamin/Cummings, Redwood City, CA.

BALTER, R., LACOURTE, S., AND RIVEILL, M. 1994. The Guide language. *Comput. J. 37,* 6, 519–530.

BAQUERO, C., OLIVEIRA, R., AND MOURA, F. 1995. Integration of concurrency control in a language with subtyping and subclassing. In *Proceedings of the USENIX COOTS Conference (COOTS '95)* (Monterey, CA).

BARGA, R. AND PU, C. 1995. A practical and modular implementation of extended transaction models. Tech. Rep. 95-004, CSE, Oregon Graduate Institute of Science & Technology, Portland, Ore.

BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA.

BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. 1988. PRESTO: A system for object-oriented parallel programming. *Softw. Pract. Exper. 18,* 8, 713–732.

BÉZIVIN, J. 1987. Some experiments in object-oriented simulation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87),* 394–405.

BIRTWISTLE, G. M., DAHL, O.-J., MYHRHAUG, B., AND NYGAARD, K. 1973. *Simula Begin.* Petrocelli Charter.

BLACK, A. P. 1991. Understanding transactions in the operating system context. *Oper. Syst. Rev. 25,* 73–77.

BLACK, A. P. AND IMMEL, M. P. 1993. Encapsulating plurality. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '93).* Lecture Notes in Computer Science, vol. 707. Springer-Verlag, New York, 57–79.

BLACK, A. P., HUTCHINSON, N., JUL, E., LEVY, H., AND CARTER, L. 1987. Distribution and abstract types in Emerald. *IEEE Trans. Softw. Eng. 13,* 1.

VAN DEN BOS, J. AND LAFFRA, C. 1991. Procol—A concurrent object-oriented language with protocols, delegation and constraints. *Acta Inf. 28,* 511–538.

BRANDT, S. AND LEHRMANN MADSEN, O. 1994. Object-oriented distributed programming in BETA. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming,* R. Guerraoui et al. Eds. Lecture Notes in Computer Science, vol. 791. Springer-Verlag, New York, 185–212.

BRIOT, J.-P. 1989. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *Proceedings European Conference on Object-Oriented Programming (ECOOP '89).* Cambridge University Press, New York, 109–129.

BRIOT, J.-P. 1996. An experiment in classification and specialization of synchronisation schemes. In *Proceedings of the Second International Symposium on Object Technologies for Advanced Software (ISOTAS '96).* Lecture Notes in Computer Science, Springer-Verlag, New York.

BRIOT, J.-P. AND YONEZAWA, A. 1987. Inheritance and synchronisation in concurrent OOP. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87).* Lecture Notes in Computer Science, vol. 276. Springer-Verlag, New York, 32–40.

BRIOT, J.-P., GEIB, J.-M., AND YONEZAWA, A., EDS. 1995. In *Proceedings of the France–Japan Workshop on Object-Based Parallel and Distributed Computation.* Lecture Notes

in Computer Science, vol. 1107. Springer-Verlag, New York.

CAMPBELL, R., ISLAM, N., RAILA, D., AND MADANY, P. 1993. Designing and implementing Choices: An object-oriented system in C++. *Commun. ACM 36,* 9, 117–126.

CAMPBELL, R. H. AND HABERMANN, A. N. 1974. The specification of process synchronisation by path expressions. In *Operating Systems,* E. Gelenbe and C. Kaiser, Eds.

CAROMEL, D. 1989. Service, asynchrony and wait-by-necessity. *J. Object-Oriented Program. 2,* 4.

CAROMEL, D. 1990. Concurrency and reusability: From sequential to parallel. *J. Object-Oriented Program. 3,* 3.

CAROMEL, D. 1993. Towards a method of object-oriented concurrent programming. *Commun. ACM 36,* 9, 90–102.

CAROMEL, D., BELLONCLE, F., AND ROUDIER, Y. 1996. The C++// System. In *Parallel Programming Using C++,* G. V. Wilson and P. Lu, Eds., MIT Press, Cambridge, MA, 257–296.

CHIBA, S. 1995. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95), Special Issue of SIGPLAN Not. 30,* 10 (Oct.), 285–299.

FINKE, S., JAHN, P., LANGMACK, O., LÖHR, K.-P., PIENS, I., AND WOLFF, T. 1993. Distribution and inheritance in the HERON approach to heterogeneous computing. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems.*

FRØLUND, S. 1992. Inheritance of synchronisation constraints in concurrent object-oriented programming languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '92).* Lecture Notes in Computer Science, vol. 615. Springer-Verlag, New York.

FRØLUND, S. 1996. *Coordinating Distributed Objects.* MIT Press, Cambridge, MA.

GARBINATO, B. AND GUERRAOUI, R. 1997. Using the strategy design pattern to compose reliable distributed protocols. In *Proceedings of the Usenix Conference on Object-Oriented Technologies and Systems (COOTS '97)* (June), S. Vinoski, Ed., Usenix.

GARBINATO, B., FELBER, P., AND GUERRAOUI, R. 1996. Protocol classes for designing reliable designing reliable distributed environments. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP '96),* (June) P. Cointe, Ed. Lecture Notes in Computer Science, vol. 1098. Springer-Verlag, New York, 316–343.

GARBINATO, B., GUERRAOUI, R., AND MAZOUNI, K. R. 1994. Distributed programming in GARF. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming,* LNCS 791, Springer-Verlag, 225–239.

GARBINATO, B., GUERRAOUI, R., AND MAZOUNI, K. R. 1995. Implementation of the GARF Replicated Objects Platform. *Distrib. Syst. Eng. J.* (Feb.), 14–27.

GOLDBERG, A. AND ROBSON, D. 1989. *Smalltalk-80. The Language.* Addison-Wesley, Reading, MA.

GUERRAOUI, R. 1995. Modular atomic objects. *Theor. Pract. Object Syst. 1,* 2, 89–99.

GUERRAOUI, R., CAPOBIANCHI, R., LANUSSE, A., AND ROUX, P. 1992. Nesting actions through asynchronous message passing: The ACS protocol. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '92).* Lecture Notes in Computer Science, vol. 615. Springer-Verlag, New York, 170–184.

GUERRAOUI, R., NIERSTRASZ, O., AND RIVEILL, M., EDS. 1994. *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming.* Lecture Notes in Computer Science, vol. 791. Springer-Verlag, New York.

GUERRAOUI, R., ET AL. 1996. Strategic research directions in object oriented programming. *ACM Comput. Surv. 28,* 4, 691–700.

GUNASEELAN, L. AND LEBLANC, R. J. 1992. Distributed Eiffel: A language for programming multi-granular distributed objects. In *Proceedings of the Fourth International Conference on Computer Languages.* IEEE Computer Science Press, Los Alamitos, Calif.

HALSTEAD, R. H. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7,* 4.

JÉZÉQUEL, J.-M. 1993. EPEE: An Eiffel environment to program distributed-memory parallel computers. *J. Object-Oriented Program. 6,* 2.

JUL, E., LEVY, H. M., HUTCHINSON, N. C., AND BLACK, A. P. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst. 6,* 1.

KAFURA, D. G. AND LEE, K. H. 1990. ACT++: Building a concurrent C++ with actors. *J. Object-Oriented Program. 3,* 1.

KALÉ, L. V. AND KRISHNAN, S. 1993. Charm++: A portable concurrent object-oriented system based on C++. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA '93), ACM SIGPLAN Not. 28.*

KARAORMAN, M. AND BRUNO, J. 1993. Introducing concurrency to a sequential language. *Commun. ACM 36,* 9, 103–116.

KAY, A. 1969. The reactive engine. Ph.D. Thesis, University of Utah.

KICZALES, G., ED. 1994. Foil for the workshop on

open implementation. Available at http://www.parc.xerox.com/PARC/spl/eca/oi/workshop-94/foil/main.html.

KICZALES, G., DES RIVIERES, J., AND BOBROW, D. 1991. *The Art of the Meta-Object Protocol,* MIT Press, Cambridge, MA.

LEA, D. 1997. *Concurrent Programming in Java.* Addison-Wesley, Reading, MA.

LEA, R., JACQUEMOT, C., AND PILLEVESSE, E. 1993. COOL: System support for distributed programming. *Commun. ACM 36,* 9, 37–47.

LEHRMANN MADSEN, O., MØLLER-PEDERSEN, B., AND NYGAARD, K. 1993. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, Reading, MA.

LIEBERMAN, H. 1987. Concurrent object-oriented programming in Act 1. In *Object-Oriented Concurrent Programming,* A. Yonezawa and M. Tokoro, Eds., Computer Systems Series, MIT Press, Cambridge, MA, 9–36.

LISKOV, B. AND SHEIFLER, R. 1983. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5,* 3.

LÖHR, K.-P. 1992. Concurrency annotations. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA '92), ACM SIGPLAN Not. 27,* 10.

LÖHR, K.-P. 1993. Concurrency annotations for reusable software. *Commun. ACM 36,* 9, 81–89.

LOPES, C. V. AND LIEBERHERR, K. J. 1994. Abstracting process-to-function relations in concurrent object-oriented applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '94).* Lecture Notes in Computer Science, vol. 821. Springer-Verlag, New York, 81–99.

MAES, P. 1987. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87), ACM SIGPLAN Not. 22,* 12, 147–155.

MAFFEIS, S. 1995. Run-time support for object-oriented distributed programming. Ph.D. Dissertation, Universität Zürich.

MASUHARA, H., MATSUOKA, S., ASAI, K., AND YONEZAWA, A. 1995. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95), ACM SIGPLAN Not. 30,* 10, 300–315.

MATSUOKA, S. AND YONEZAWA, A. 1993. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming,* G. A. Agha et al., Eds., MIT Press, Cambridge, MA, 107–150.

MAZOUNI, K., GARBINATO, B., AND GUERRAOUI, R. 1995. Building reliable client-server software using actively replicated objects. In *Proceedings of TOOLS Europe '95,* Prentice-Hall, Englewood Cliffs, NJ, 37–53.

MCAFFER, J. 1995. Meta-level programming with CodA. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '95).* Lecture Notes in Computer Science, vol. 952. Springer-Verlag, New York, 190–214.

MCHALE, C. 1994. Synchronisation in concurrent, object-oriented languages: Expressive power, genericity and inheritance. Ph.D. dissertation. Dept. of Computer Science, Trinity College, Dublin, Ireland.

MEYER, B. 1991. *Eiffel: The Language,* Prentice-Hall, Englewood Cliffs, NJ.

MEYER, B. 1993. Systematic concurrent object-oriented programming. *Commun. ACM 36,* 9, 56–80.

MOSS, J. E. B. AND KOHLER, W. H. 1987. Concurrency features for the Trellis/Owl language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87).* Lecture Notes in Computer Science, vol. 276. Springer-Verlag, New York.

MOWBRAY, T. J. AND ZAHAVI, R. 1995. *The Essential CORBA: System Integration Using Distributed Objects.* John Wiley & Sons and The Object Management Group, New York.

NICOL, J., WILKES, T., AND MANOLA, F. 1993. Object-orientation in heterogeneous distributed computing systems. *IEEE Comput. 26,* 6, 57–67.

OKAMURA, H. AND ISHIKAWA, Y. 1994. Object location control using metalevel programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '94).* Lecture Notes in Computer Science, vol. 821. Springer-Verlag, New York, 299–319.

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

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

PAPATHOMAS, M. 1989. Concurrency issues in object-oriented programming languages. In *Object-Oriented Development,* D. C. Tsichritzis, ed. Centre Universitaire d'Informatique, Université de Genève, Geneva, Switzerland.

PAPATHOMAS, M. 1995. Concurrency in object-oriented programming languages. In *Object-Oriented Software Composition,* O. Nierstrasz and D. Tsichritzis, Eds. Prentice-Hall, Englewood Cliffs, NJ, 31–68.

PARRINGTON, G. D. AND SHRIVASTAVA, S. K. 1988.

Implementing concurrency control in reliable distributed object-oriented systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '88)*. Lecture Notes in Computer Science, vol. 322. Springer-Verlag, New York, 234–249.

ROZIER, M. 1992. Chorus. In *Proceedings of the Usenix International Conference on Micro-Kernels and Other Kernel Architectures,* 27–28.

SCHILL, A. AND MOCK, M. 1993. DC++: Distributed object-oriented system support on top of OSF DCE. *Distrib. Syst. Eng. 1,* 2, 112–125.

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

SCHRÖDER-PREIKSCHAT, W. 1994. *The Logical Design of Parallel Operating Systems.* Prentice-Hall, Englewood Cliffs, NJ.

SHEFFLER, T. J. 1996. The Amelia vector template library. In *Parallel Programming Using C++,* G. V. Wilson and P. Lu, eds. MIT Press, Cambridge, MA, 43–90.

SKJELLUM, A., LU, Z., BANGALORE, P. V., AND DOSS, N. 1996. MPI++. In *Parallel Programming Using C++,* G. V. Wilson and P. Lu, eds. MIT Press, Cambridge, MA, 465–506.

STROUSTRUP, B. 1993. *The C++ Programming Language.* Addison-Wesley, Reading, MA.

STURMAN, D. AND AGHA, G. 1994. A protocol description language for customizing failure semantics. In *Proceedings of the Thirteenth International Symposium on Reliable Distributed Systems,* 148–157.

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

THOMAS, L. 1992. Extensibility and reuse of object-oriented synchronisation components. In *Proceedings of the International Conference on Parallel Languages and Environments (PARLE '92).* Lecture Notes in Computer Science, vol. 605. Springer-Verlag, New York, 261–275.

TOKORO, M., NIERSTRASZ, O. M., AND WEGNER, P., EDS. 1992. *Proceedings ECOOP '91 Workshop on Object-Based Concurrent Computing.* Lecture Notes in Computer Science, vol. 612. Springer-Verlag, New York.

WEGNER, P. 1990. Concepts and paradigms of object-oriented programming. *ACM OOPS Manager 1,* 1.

WEIHL, W. 1989. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst. 11,* 2.

WILSON, G. V. AND LU, P., EDS. 1996. *Parallel Programming Using C++.* MIT Press, Cambridge, MA.

YOKOTE, Y. 1992. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '92). ACM SIGPLAN Not. 27,* 10, 414–434.

YOKOTE, Y. AND TOKORO, M. 1987. Experience and evolution of Concurrent Smalltalk. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87). ACM SIGPLAN Not. 22,* 12.

YONEZAWA, A. AND TOKORO, M., EDS. 1987. *Object-Oriented Concurrent Programming.* Computer Systems Series, MIT Press, Cambridge, MA.

YONEZAWA, A., BRIOT, J.-P., AND SHIBAYAMA, E. 1986. Object-oriented concurrent programming in ABCL/1. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86). ACM SIGPLAN Not. 21,* 11.

YONEZAWA, A., MATSUOKA, S., YASUGI, M., AND TAURA, K. 1993. Implementing concurrent object-oriented languages on multicomputers. *IEEE Parallel Distrib. Technol.* (May).

Received August 1996; revised August 1997; accepted January 1998