Concurrency, Distribution and Parallelism in Object-Oriented Programming

Jean-Pierre BRIOT
Laboratoire d'Informatique de Paris 6, UPMC
Jean-Pierre.Briot@lip6.fr

Rachid GUERRAOUI
Département d'Informatique, École Polytechnique Fédérale de Lausanne
guerraoui@di.epfl.ch

Klaus-Peter LÖHR
Institut für Informatik, Freie Universität Berlin
lohr@inf.fu-berlin.de

Technical Report B–97–14
December 1997

Abstract

This paper aims at classifying and discussing the various ways along which the "object" paradigm is used for concurrent systems. We distinguish the *library* approach, the *integrative* approach and the *reflective* approach. The library approach applies object-oriented concepts, as they are, to structure concurrent systems through libraries. The integrative approach consists in merging concepts such as object and activity, message passing and transaction. The reflective approach closely integrates protocol libraries and object-oriented languages. We discuss and illustrate each of these approaches and point out their complementary levels and goals. We will also make a careful distinction between the notions of *concurrency* on the one hand - referring to the non-sequential semantics of a program - and *parallelism* and *distribution* on the other hand - referring to the actual implementation of a concurrent system.

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

Freie Universität Berlin
Institut für Informatik
Takustraße 9
D–14195 Berlin, Germany
http://www.inf.fu-berlin.de

1 Introduction

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

Most object-oriented programming languages do have some concurrency extensions or libraries, and almost every new architectural development in the distributed system community is, to some extent, object-oriented. For instance, both the Open Distributed Processing (ODP) and the Object Management Group (OMG) standardization initiatives for heterogeneous distributed computing are based on object concepts.

As a result, many object-oriented models, languages and architectures for concurrent, parallel and distributed systems have been proposed and described in the literature. Towards a better understanding and evaluation of these proposals, this paper addresses the question of how object orientation and concurrency can be accommodated in a common framework. Rather than presenting an exhaustive study of relevant systems, this paper aims at extracting, classifying and discussing the various ways the object paradigm is used in those systems.

1.1 Three approaches to concurrency

A coarse classification identifies three approaches: the *library* approach, the *integrative* approach and the *reflective* approach.

Using the *library* approach, concurrency features are added to a sequential object-oriented model through libraries. For example, library classes for processes, semaphores, messages etc. can be provided. The language proper is not changed.

The *integrative* approach aims at unifying concepts of concurrency and of object orientation, either by extending an existing (sequential) language or by designing a new language. Concepts of object orientation and concurrency are merged. For example, merging the notions of *process* and *object* gives rise to the notion of *active object*, and this in turn implies a unification of the notions of *message passing* and *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.

The *reflective* approach integrates protocol libraries within an object-oriented programming language. The idea is to separate the application program from the various aspects of its implementation and computation contexts (models of computation, communication, distribution etc.), which are described in terms of *meta-program(s)*. Reflection may also deal with resource management, such as load balancing and time dependencies, and describe it with the full power of a programming language. The reflective approach may be considered as a "bridge" between the two previous approaches as it helps to transparently integrate various computing protocol libraries within a programming language/system. Moreover, it helps to combine the two other approaches, by making explicit the separation of, and the interface between, their respective levels (i.e.: the integrative approach for the end user, and the library approach for developing and customizing the system).

Although these approaches seem to compete at a first glance, they are in fact complementary: The *library* approach is oriented towards systems programmers and aims at identifying required concurrency concepts and casting them into library classes. The *integrative* approach is oriented towards application programmers and aims at defining a high-level programming language supporting a powerful concurrent object model. The *reflective* approach does not take sides. Its main goal is to provide the basic infrastructure to enable (dynamic) system customization with minimal impact on the application programs. The success of a reflective system relies both on a high-level programming language and on a rich library of concurrency abstractions.

1.2 Parallel and distributed implementation of concurrency

We will make a careful distinction between *concurrent, parallel* and *distributed.* There are different ways of running a concurrent program on an execution platform. The program may be executed on a uniprocessor, e.g., using a threading system, or on a parallel computer, or even on a network of computers. Thus, while concurrency is a *semantic* property of a program, parallelism and distribution pertain to its *implementation*, as determined by the compiler and other systems software. Regarding distribution, the notion of *distribution transparency* is often used to emphasize the fact that the distributed implementation is not reflected in the program text.

We will therefore be reticent with notions such as “parallel program” or “distributed program” and rather speak of *parallel and/or distributed implementation* of a concurrent language or program. We feel that only if the underlying parallel and/or distributed system architecture is somehow reflected in the program text, defeating transparency, the program can justifiably be called “parallel” or “distributed”.

It should be kept in mind that distribution does not necessarily imply concurrency: a purely sequential program may be executed across machine boundaries, using remote procedure calls. A similar situation is found with client/server systems: while a server may or may not be concurrent, its clients rarely are; only when we view a server and its clients as one system we see a concurrent system, operating in a distributed fashion.

1.3 Active vs. passive objects

What kind of relationship exists between the notions of *process* (as used in traditional concurrent programming) and *object*? There are several possible answers to this question, ranging from ”they are unrelated” to ”they are just synonyms”. For swift introduction of concurrency into an existing object-oriented language the first answer may seem the natural one. An Ada programmer, however, knowing that an Ada task is both a process and an invokable object, might prefer the second answer.

Transplanting the Ada task concept into the object-oriented world gives rise to *active objects*, as opposed to the common ”passive” objects used in purely sequential object-oriented programming. As we will see, one of the distinguishing features of an active object, its independent activity, actually comes in different flavors. For example, the object may start its activity spontaneously (as in Ada), or it may have to be triggered by an invocation; it may feature one thread of control (as in Ada), or it may have several threads.

Note that in principle the particular choice of an active object model will be independent of whether that model is implemented in a library or employing special language constructs or using reflection. In reality, the more interesting active object models are found in integrative systems, for reasons to become clear below.

1.4 Previous work

The reader is assumed to be familiar with traditional concurrency concepts as described, e.g., in [Andrews 91]. It should also be kept in mind that object-based concurrent programming is a well-established discipline which is supported by many languages. *Ada* is a well-known example [Ada 83] [Barnes 97]. *SR*, although less known, features versatile and powerful constructs [Andrews/Olsson 93]. *Emerald* has become known for its distribution support [Jul et al. 88].

Combining concurrency with object orientation proper, i.e., including inheritance, has been the subject of many research projects since 1985. Several new language designs, representing the integrative approach, are discussed and compared in [Papathomas 89] and [Papathomas 95]. An early book featuring different articles on concurrent object-oriented programming is [Yonezawa/Tokoro 87], a more recent one is [Agha et al. 93]. For workshops on the subject the reader is referred to [Agha et al. 89] [Agha et al. 91] [Tokoro et al. 92] [Guerraoui et al. 94] [Ciancarini et al. 95] [Briot et al. 95].

Much effort has recently been put into C++ variants and libraries for parallel processing. For a collection of articles about different projects on C++ and parallelism see [Wilson/Lu 96].

As mentioned above, our treatment of concurrent object-oriented programming is not meant as an exhaustive study of the relevant programming languages. A fairly complete survey (as of 1995) focusing on parallelism and including an annotated bibliography can be found in [Philippsen 95a,b].

2 The Library Approach

The basic idea of the library approach is to use encapsulation and abstraction, and possibly also classes and inheritance, as structuring tools for concurrency mechanisms imported from a certain execution platform. This should facilitate the construction of concurrent systems using an object-oriented methodology and a given, sequential, object-oriented language.

To illustrate the approach, we survey the following examples: (1) the Smalltalk-80 programming language and environment, where a basic and simple object concept is uniformly applied to model and structure the whole system through class libraries, including concurrency and distribution aspects; (2) the Eiffel programming language, for which several libraries have been proposed to address concurrency; (3) C++, whose widespread use has resulted in a proliferation of concurrency libraries. We will also briefly mention concurrency support in object-oriented operating systems such as Choices and Peace.

2.1 Smalltalk

Smalltalk is often considered as one of the "purest" examples of object-oriented languages [Goldberg/Robson 89]. This is because its "motto" is to have only a few concepts (object, message passing, class, inheritance) and to *apply them uniformly* to any aspect of the language and environment. One consequence is that the language is actually very simple; its richness comes from its set of class libraries. They describe and implement various programming constructs (control structures, data structures etc.), internal resources (messages, processes, compiler etc.), and a sophisticated programming environment with integrated tools (browser, inspector, debugger etc.).

2.1.1 Standard class libraries

In Smalltalk, even basic control structures, such as loop and conditional, are not primitive language constructs, but just standard methods of standard classes, which make use of the generic invocation of message passing. They are based on booleans and execution closures - *blocks*. Blocks, represented as instances of class `BlockContext`, are essential for building various control structures that may be extended as required. They are also the basis for *multi-threaded concurrency* through processes. Standard class `Process` describes their representation; the associated methods implement process management (suspend, resume, adjust priority etc.). The process scheduler is the single instance of the class `ProcessorScheduler`. Smalltalk processes are implemented using coroutines or threads.

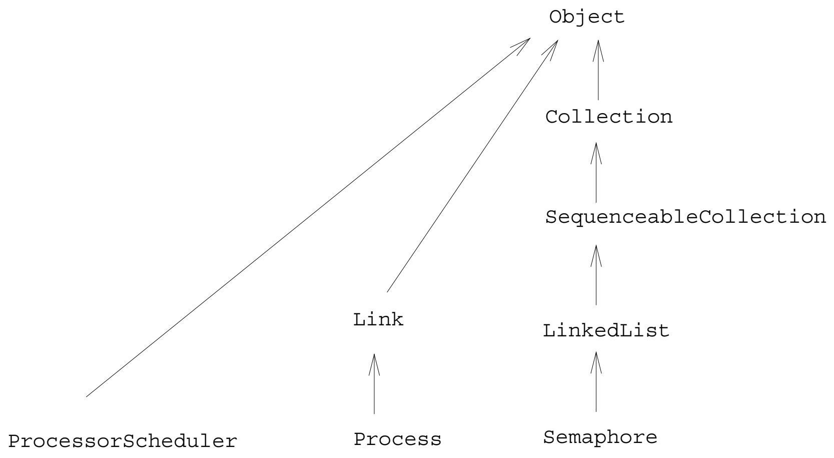

Figure 1: Concurrency in Smalltalk - The "inherits from" relationship

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

Smalltalk also offers libraries for remote communication using sockets and RPC, and for storage and exchange of object structures, supporting marshaling, persistence and transactions. The *Binary Object Streaming Service (BOSS)* library provides a basic support for building distribution mechanisms.

2.1.2 Extending class libraries

Due to Smalltalk's uniform approach, concurrency concepts and mechanisms are well encapsulated and organized in a class hierarchy. Thus, they are better understandable than if they were just a set of primitives in the programming language. It is also relatively easy to build more sophisticated abstractions on top of the basic standard library.

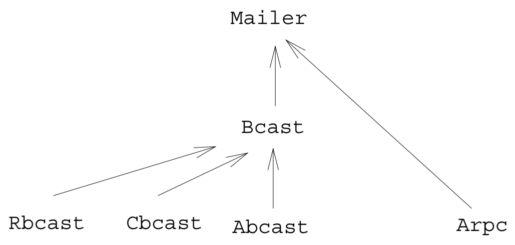

Figure 2: Communication in GARF - The "inherits from" relationship

Inheritance has been used extensively to structure various services for concurrent programming. An example is the *Simtalk* platform [Bézivin 87], which implements and classifies various synchronization and simulation abstractions (Hoare monitors, Kessels monitors, pessimistic or optimistic simulation objects etc.) on top of Smalltalk standard abstractions/classes. For distributed and fault-tolerant programming abstractions, an example is the GARF project [Garbinato et al. 94], in which two complementary class hierarchies have been developed for various communication models (point-to-point, multicast, atomic multicast etc.) (see Figure 2) and object models (monitor, persistent, replicated etc.). The HP *Distributed Smalltalk* product provides a set of distributed services following the OMG CORBA standard [OMG 95], also implemented as Smalltalk-80 class libraries.

In a similar approach for the *Beta* programming language [Lehrmann Madsen et al. 93], a library of classes (named "patterns" in Beta) for distributed programming has been developed [Brandt/Lehrmann Madsen 94]. For instance, class `NameServer` represents a name server which maps textual object names to physical references. Class `ErrorHandler` manages partial errors/faults of a distributed system. This approach enables the programmer to add distributed features to a given sequential/centralized program without changing the program logic, i.e., through additions rather than changes.

2.2 Eiffel

While Smalltalk is untyped (or at least not statically typed), *Eiffel* [Meyer 91] is a language with static typing. Originally designed as a sequential object-oriented language, it has been extended towards concurrency in different ways. Language extensions proper will be considered later. Here we concentrate on the library approach to Eiffel concurrency.

Existing concurrency libraries for Eiffel demonstrate that there are fundamentally different ways to model an independent activity (process, task, thread etc.) as an object. We observe that two kinds of data belong to a process: application data and management data; they can be either

1. *separated:* a process object encapsulates data and operations just for management of the process; the executed code is responsible for its application data and any inter-process communication;
2. *integrated:* a process object encapsulates both management and application code and data; consequently, all process management and inter-process communication is performed through the object's interface; we thus have an *active object* in the sense of 1.3.

Be careful not to confuse the object-process integration leading to active objects with what we have defined as the integrative approach - they are independent of each other. In fact, as we will see later, either of the three approaches - library, integrative, reflective - can accommodate both object-process separation and object-process integration.

Separation is the basis of the Smalltalk solution. A corresponding variant for Eiffel is described in [Colin/Geib 91]. We will discuss an integrated solution, presented in [Karaorman/Bruno 93] for Eiffel-2, the precursor of the current version of Eiffel.

If a class inherits from the library class `Concurrency` its objects are active objects. The behaviour of an active object resembles that of an Ada task, its operations corresponding to the Ada task entries. Essentially, the object executes a loop, checking for and accepting pending invocations. As opposed to Ada, service execution is *always* asynchronous; results, if any, are returned using *futures* (which are similar to the *promises* mentioned in 2.1.1) [Halstead 85]. The programmer of an active object class is responsible for redefining the routine `scheduler` inherited from `Concurrency`; `scheduler` must contain the loop that controls the behaviour of the object.

A simplified example is shown in Figure 3. Active `Printer` objects are capable of asynchronously executing `print` requests. Note that `Printer` does not export `print`. `Concurrency` exports `remote-invoke` (among others), and it is this operation that has to be called by a client as shown. (`remote-invoke` is implemented as sending a message which is then picked up by a separate process executing the never-ending `scheduler`).

We see that this technique does not achieve a fully transparent solution that would allow the same syntax for invoking active and passive objects (`p.print(arg)`). So although the system achieves more than just supporting passive `Process` objects, it also exhibits problems with the library approach. The subject is discussed in detail in [Karaorman/Bruno 93]; a methodology for alleviating the problems is suggested.

```
CLASS Printer
INHERIT Concurrency REDEFINE scheduler
FEATURE .....  -- any attributes are declared here

        print(filename: String) IS ..... END -- print;

        scheduler IS LOCAL fn: String
        DO FROM get_request UNTIL false
           LOOP current_request := request_queue.remove;
                fn ?= current_request.parameters.item(1);
                print(fn); .....  END -- loop
        END -- scheduler
END -- Printer


.....  -- in a client class
p: Printer;
...
p.remote_invoke("print", args);
...
```

Figure 3: Definition and usage of a class for active objects

The designers of the system have decided to implement active objects as Unix heavy-weight processes rather than light-weight threads. So invocation of an active object always involves Unix inter-process communication. This makes the approach unsuitable for medium- to small-grain concurrency and highly parallel computation. It is readily usable, though, for physical distribution on platforms that support remote process spawning.

Highly parallel computation is supported by the *EPEE* system [Jézéquel 93b]. EPEE follows the SPMD (single program, multiple data) approach to data parallelism: large data aggregates (such as, e.g., 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. The essentials of EPEE are:

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

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

EPEE's ideas are close to other object-oriented approaches to massive parallelism, notably those of Concurrent Aggregates [Chien 93a] and Charm++ [Kalé/Krishnan 93]. We will come back to these in section 3.5. EPEE is not as elegant, but this may be the price that had to be paid for not changing the language.

2.3 C++

As opposed to Smalltalk and Eiffel, C++ [Stroustrup 93] is not genuinely object-oriented. It is an object-oriented extension of C, a language originally designed for systems programming. Thus, it 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.

This implies that combining C++ with concurrency libraries is more than a marriage of convenience. As explained in 1.1, the systems programmer needs flexibility and therefore prefers libraries to built-in features. He also likes to exploit the low-level concurrency mechanisms offered by the underlying execution platform. As the library approach allows for any functionality of a given platform to be wrapped in C++ functions or classes, it is not surprising that there is a wide variety of concurrency mechanisms cast in C++ libraries. In fact, any programmer can readily build wrappers for concurrency mechanisms from her or his favourite platform.

2.3.1 Threading libraries

Class libraries can be built for all kinds of process concepts, heavy-weight or light-weight, and for their corresponding synchronization mechanisms. We have seen how concurrency can be added to Eiffel by providing a library class `Concurrency` which is implemented using a heavy-weight Unix process. Many concurrent programs, however, are conveniently implemented using a threading system (e.g., network servers, interactive programs, parallel programs). So it is important to look into object-oriented threading libraries.

Is a thread an object? While 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, depending on how the activity of a thread object is described:

1. A thread is an instance of a subclass of `Thread`, and its activity is described by the *constructor* of the subclass. This is akin to the Simula approach to coroutines: the body of a coroutine class describes both initialization and activity of a coroutine object.

2. Again, a thread is an instance of a subclass of `Thread`, but its activity is described by overriding a *special method* (similar to the `scheduler` routine in 2.2).
3. A thread is an instance of `Thread`, and its activity is described by a function that is passed as a *parameter* to the constructor or a special method.

In all these approaches, creating a thread object spawns a new thread. Note that the lifetime of its activity may be shorter than the lifetime of itself (as an object). Also note that although a thread object is "active" in a way (because the thread is executing some code), it is *not* an *active object*.

An example of approach 1 is found in the *coroutine library* part of Sun's C++ library [Sun 95]. The library offers a class `task` (not `Thread`). A `task` object is implemented as a coroutine, i.e., with non-preemptive scheduling. There is also a class `Interrupt_handler` that allows to catch Unix software interrupts (signals). Typical operations on tasks are `result()` (wait for termination), `rdstate()` (get state) etc. Synchronization is supported by low-level wait operations and by object queues.

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

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

Figure 4: Typical scenario for thread objects: customized Sun C++ `task` object

Figure 4 shows a fragment of a simple program using the coroutine library. The main program, by declaring the object `p`, creates a task which executes the `producer()` constructor. There is no interaction between parent and child task, except that the child terminates with producing a result, which is picked up by the parent.

An example of the third approach sketched above is found in *PRESTO*, a system for parallel programming on a multiprocessor [Bershad et al. 88]. 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.

The function to be executed is passed to the constructor in *DC++* [Schill/Mock 93], a system for distributed execution of C++ programs on top of DCE, the OSF

Distributed Computing Environment [OSF 94] [Rosenberry et al. 93]. While DC++ focuses on distribution it does offer 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.

An example that is typical for C++, exploiting overloading and templates, is the threading library of the *ACE* system [Schmidt 95]. ACE stands for Adaptive Communications Environment; 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.

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

```
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: Threads are not objects in ACE

Threads are handled on a very low level of abstraction in ACE. There does exist a class `Thread`, but this is just a package of static functions such as `spawn, join, yield` etc., abstracting from the idiosyncrasies of the threading functions of POSIX, Solaris and Win32 (Figure 5). Another class, `Thread_Manager`, does serve the purpose of creating and using thread manager objects; they are responsible for managing groups of threads, spawning new members, disposing of a thread when it terminates etc. But there is no class that would resemble Smalltalk's `Process` or the `task` from Figure 4.

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

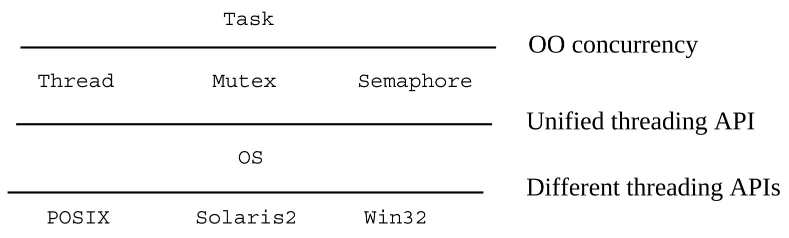

Figure 6: Architecture of the ACE concurrency library

Figure 6 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.

2.3.2 Distribution support

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 message passing mechanisms such as ports or sockets (e.g., ACE supports Unix socket objects).

This is unfortunate because the object invocation paradigm is lumped together with the message passing paradigm of distributed computation. In many cases it is also not necessary because *remote invocation* would be the mechanism of choice: it hides message passing and achieves distribution transparency in a genuinely object-oriented program. Remote object invocation, the object-oriented analog to remote procedure call, is a technique that does not require either changes in the language or explicit usage of libraries.

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

The most prominent platform for object-oriented distributed computing is CORBA [OMG 95] [Mowbray/Zahavi 95]. Although CORBA is language-independent, the bulk of support available for CORBA today is geared towards C++. Concurrency is not a central issue in CORBA. But implementors of server objects may of course be confronted with the need for concurrency control. Therefore, an object transaction service and a non-transactional concurrency control service are provided.

2.3.3 Parallel Computing

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 the programs described in 2.3.1 on a shared-memory multiprocessor, as mentioned above for PRESTO. This produces functional parallelism, but no data parallelism.

We have seen that EPEE uses Eiffel objects to represent fragments of data aggregates (also called collections) for data-parallel programming. A different way of accommodating the notions of object and aggregate is to simply identify them. A popular technique is to provide library classes for certain kinds of aggregates, hiding fragmentation and communication. This, of course, leaves the programmer with a limited set of predefined aggregate classes. But still, a good degree of flexibility can be achieved in C++ by using templates. The Amelia Vector Template Library (AVTL) [Sheffler 96] is an example of library support for parallel processing of vectors.

While an approach like AVTL is tailored towards a specific class of applications, it has the advantage of hiding communication from the programmer. If we are willing to pay the price of low-level message-based programming, unlimited flexibility is achieved by libraries that connect to a communication platform, e.g., MPI [Skjellum et al. 96]. Libraries of this kind can be seen as the "distributed-parallel" equivalent to threading libraries as described above.

2.3.4 Object-oriented operating systems

Choices [Campbell et al. 93] is a *generic* operating system: it was designed to be easily ported onto various machines, but also to be adjustable to various characteristics of both hardware, resources, and application interfaces, such as file format, communication network, and memory model (shared or distributed). An object-oriented methodology is presented together with the system, both for the design of distributed applications and for the design of new extensions to the Choices kernel.

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

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

Distributed memory multicomputers are the target of the *Peace* parallel operating system [Schröder-Preikschat 94]. Peace is actually a *family* of object-oriented operating systems. Its components, implemented in C++, can be configured in different ways, in order to fit different hardware platforms and offer varying functionality.

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

The basic unit of concurrent execution, the *thread*, is introduced through a series of abstractions. Most threads are made up from two objects, of classes `native` and `thread`, respectively. The class `native` describes the kernel-level part of the thread, `thread` refers to the user-level part. An application program can declare a subclass of `thread`, say, `custom`, redefining the method `action()` inherited from `thread`. Creating a `custom` object causes the creation of a thread that executes the redefined `action()`. The situation is not unlike the one described in 2.3.1.2, with the same caveat: a thread is *not* an active object (contrary to the terminology in Peace) because there is no provision for communicating with the thread through the `custom` interface.

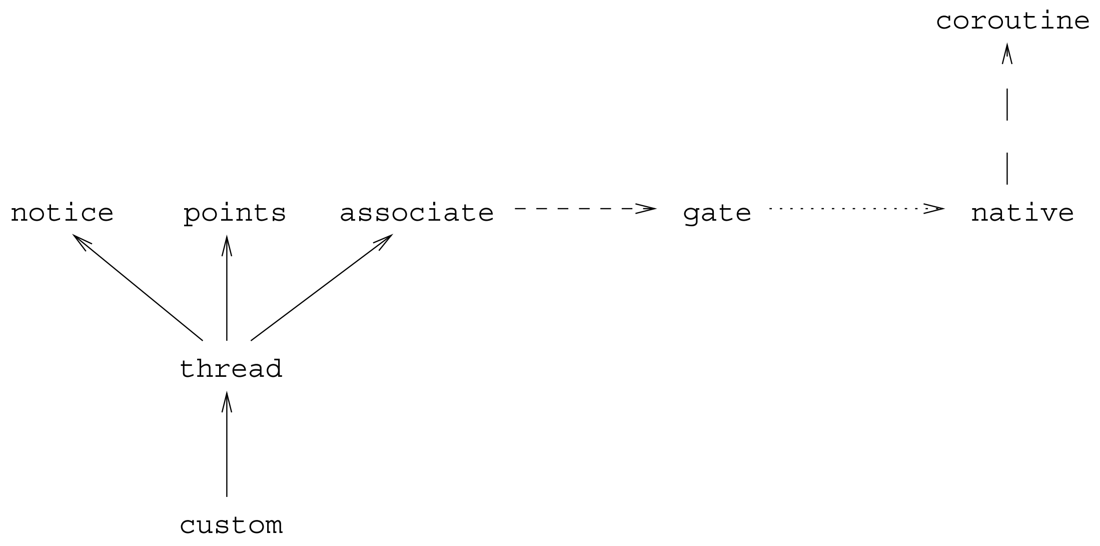

Figure 7: Small excerpt from the architecture of Peace (simplified). “inherits from” proceeds *bottom-up*, and “uses” proceeds *left-to-right*.

Figure 7 shows an excerpt from the architecture of Peace. There is a sequence of abstractions, implemented through inheritance, that leads from `coroutine` to `native`. The `associate` part of a user-level thread refers to a `gate` object that knows the unique identifier of the kernel-level thread.

Threads interact either via shared objects or, if located in different address spaces, through message passing. Intra-address-space thread synchronization is achieved via *event counters* which are supported by `notice`. A variety of message passing mechanisms is supported.

2.4 In search for standard abstractions

The main issue underlying the library approach is the design and implementation of adequate abstractions on top of which various higher-level concurrency 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 (as, e.g., the `Guard` class of ACE). Classification and specialization mechanisms, as offered by object-oriented programming, are then appropriate to organize such a library/hierarchy of abstractions, as for instance in the the Simtalk platform (Sect. 2.1). Peace is a typical example for an extremely careful design of a hierarchy of thread abstractions.

An example of developing concurrency abstractions complementary to program abstractions can be found in the *Demeter* environment [Lopes/Lieberherr 94]. 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 between these three components help with the development of generic and reusable programs.

A similar study of abstractions for distributed programming has been proposed in [Black 91] 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 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]. Both the Venari [Wing 94] and the Phoenix [Guerraoui/Schiper 95] projects aim at defining various transactional models from a set of minimal abstractions.

2.5 Evaluation of the library approach

In summary, the library approach aims at increasing the flexibility, yet reducing the complexity, of concurrent computing systems by structuring them as libraries

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

of classes. Each aspect or service is represented by an object. Such modularity and abstraction objectives are very important because concurrent computing systems are rather complex and ultimately use very low-level mechanisms, such as processor switching and network communication. Furthermore, such systems are often developed by teams of programmers; having separate modules with well-defined interfaces is of prime importance in such a context. Finally, the difficulty with maintaining and extending Unix-like systems is mainly due to low modularity and poor abstractions.

Although progress is being made towards that direction, as noted above, it is still too early to come up with a standard class library for concurrent programming. It may even be a red herring. We need a good knowledge of the minimal mechanisms and their interaction; in addition, we need a consensus about the desirable extensions and how to structure them. Different technical communities are involved: people from programming languages, operating systems, distributed systems, database sytems and, last but not least, real-time systems. The fact that the semaphore abstraction became a standard primitive for synchronization is just a tiny grain of hope that a generally accepted library of concurrency abstractions might emerge.

3 The Integrative Approach

We have seen that the library approach to concurrent object-oriented programming starts out from a given concurrency platform such as an operating system or a user-level threading package. The low-level concurrency mechanisms offered by the platform are cast into objects. Wherever possible, inheritance is used as an aid in structuring a concurrency library.

This bottom-up approach, although flexible, is not attractive for the application programmer. If we follow the object paradigm all the way down from analysis through modeling through design to implementation, we would like to stay on a problem-oriented level. Now the library approach does help with the object-oriented structuring of low-level mechanisms like threads, semaphores, messages etc.; but those objects are not hidden: they are kept separate from the objects structuring the application. In other words, there is no integration between the two tasks the programmer faces: programming with application objects *and* managing concurrency, also with objects, but *not the same objects!*

Furthermore, programming may become cumbersome when using libraries, and concurrency management may obscure the application logic. For instance, subclasses of the `Concurrency` class mentioned in 2.2 give rise to active objects. But in programming such a class we are forced to manage explicit acceptance of invocations. It would be more attractive to have a solution where classes for active objects are not much different from regular classes.

In summary, rather than keeping application logic and concurrency management separate, we should strive for a solution that offers a unified object model to the programmer. This cannot be achieved through libraries. An *integrative* approach is necessary where concurrency semantics are built into the language, either from the beginning or by extending a given sequential language.

3.1 Degrees of object-process integration

The concept of *active object* which integrates the notions of object and process is appealing and natural: it immediately occurs to us when we model reality. But it is not the only way of incorporating concurrency into an object-oriented language. There is a variety of ways, exhibiting *different degrees* of object-process integration:

1. *No integration:* The language supports independent activities (called processes, threads, tasks or whatever), but these are *not objects.* All objects are passive, and activities interact via shared (and properly synchronized) objects. (Examples: Modula-3 [Harbison 92], Guide [Decouchant et al. 91] [Balter et al. 94], Beta [Lehrmann Madsen et al. 93], pSather [Feldman et al. 93]).
2. *Poor integration:* a) Process objects do exist, but they are a species *different from class objects.* They are instantiated from certain process templates which do not participate in inheritance. (Example: Concurrent C++ [Gehani/Roome 88].) Or b) process objects are instantiated from a built-in `Thread` class, but have no application-specific interface; operations on processes are stop, start, reschedule etc. The semantics is very close to that of the library approach described in 2.3.1. (Example: Java [Arnold/Gosling 96] [Lea 97].)
3. *Partial integration:* There are two kinds of objects, *active* and *passive.* Inheritance among their classes is possible, but subject to certain restrictions. (Examples: Eiffel// [Caromel 93], Charm++ [Kalé/Krishnan 93].)
4. *Full integration:* All objects are potentially active, and there are no restrictions regarding inheritance. (Examples: Pool [America/van der Linden 90], CEiffel [Löhr 93].)

This classification is a refinement of the simple distinction made in the context of the library approach in 2.2: separated vs. integrated handling of management data and application data of a process object. Note that integration in the sense of 2.2 is fully realized in 2a), 3 and 4.

The integrative approach tries to integrate into the application objects not only processes but also synchronization. For *passive objects* this is similar to rejecting *synchronization objects* (e.g., semaphores) and instead using *synchronized objects* (e.g., monitors) in traditional concurrent programming. *Active objects* use synchronized invocation similar to the Ada rendezvous. In both cases, however, it is not obvious how synchronization relates to inheritance. It turns out that inheritance may give rise to a phenomenon called *inheritance anomaly.* We will return to this later.

3.2 Passive objects

There are rare cases where allowing several concurrent activities to access a given passive object cannot do any harm. Examples are read-only objects or very small objects where the operations are implemented as indivisible machine instructions (e.g., increment an integer variable). In most cases, however, shared objects in a concurrent environment have to be *synchronized.* It is helpful to distinguish between different sources of the need for synchronization:

- *specification* of the object: *condition synchronization*;
- *implementation* of the object: *exclusion synchronization*;
- *scheduling* of the operations (if different from built-in scheduling).

We will discuss each of these in turn.

3.2.1 Condition synchronization

The specification of, e.g., job queues would indicate that the operation to `remove` a job from a `Queue` can only be executed under the precondition that the queue is not empty. Thus, acceptance of a `remove` invocation has to be delayed when the queue is empty.

A problem-oriented solution to this synchronization problem is to use a declarative *guard*, i.e., a concrete version of the abstract precondition, expressed as a predicate over the object's data representation (Figure 8, upper part, with ad-hoc syntax). Guards are well-known in traditional concurrent programming and can be found, e.g., in Ada, SR and Orca [Bal et al. 92]. Guards achieve the desired integration because they do not require any synchronization statements in the implementation of the object's operations. Activities are blocked or woken up implicitly. The price that has to be paid for this is performance; explicit operations such as signalling a monitor event or a semaphore are more efficient.

```
CLASS Queue;
VAR length: Natural;
OPERATION remove: Item;
          WHEN length>0
          END ..... 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 8: Guarded `remove` operation in class `Queue`

In the language *Guide* the guards are gathered in a central location of the class, called the control clause (see Figure 8, lower part). A guard can refer not only to the object's state and the operation's parameters, but also to *counters* which indicate certain numbers connected with operations [Robert/Verjus 77]; e.g., `completed(op)` is the number of completed executions of `op`.

Referring to counters has the advantage of representation independence but may have drawbacks when it comes to inheritance. The guards in the control clause are inherited and can be redefined if necessary. Figure 9 shows a subclass of `Queue`, `ExtendedQueue`: a new operation `delete` for deleting the last element has been added. Notice that it does not suffice to add a new guard for `delete` to the control clause. Annoyingly, we have to redefine the `remove` guard although `remove` itself is not redefined. This anomaly is one example of a species of phenomena called *inheritance anomaly*. We will encounter more examples below. A thorough treatment of inheritance anomalies can be found in [Matsuoka/Yonezawa 93].

```
CLASS ExtendedQueue INHERIT Queue
OPERATION delete;
          WHEN length>0
          BEGIN ..... END; END ExtendedQueue.


---------------------------------------------------------------------

CLASS ExtendedQueue SUBCLASS OF Queue IMPLEMENTS ExtendedQueueType IS
METHOD  delete;
        BEGIN ..... END delete;

CONTROL remove: completed(append)>completed(remove)+completed(delete);
        delete: completed(append)>completed(remove)+completed(delete);
END ExtendedQueue.
```

Figure 9: Inheritance anomaly when using counters

The expressive power of counters is not very high. Imagine we need a `Clearable-Queue`, to be derived from `Queue` by adding a `clear` operation. There is no way to express the new synchronization requirements by means of counters. Fortunately, Guide does not rely solely on counters: the guards may also refer to the object's represention. If we derive our subclass from another version of `Queue` where the control clause reads `remove: length>0`, we arrive at the trivial solution shown in Figure 10. (A solution for the above `ExtendedQueue` is also very simple.) The lesson to be learned is: try to avoid using counters for condition synchronization.

Another high-level approach to condition synchronization is to use *synchronization expressions*: all information about synchronization is centralized in one expression which specifies all possible histories (= sequences of operation executions) of an object. This approach is attractive because a synchronization expression is completely independent of the object's representation; it is, in fact, part of the specification (or derivable from it).

Synchronization expressions are akin to *path expressions* [Campbell/Habermann 74]. Because the original path expressions lack expressive power, later versions have been extended with predicates. A predicate may again refer to *counters* as mentioned above [Andler 79], or it may be a guard referring to the object's state [van den Bos/Laffra 91]. The latter version obviously defeats what makes path expressions attractive in the first place.

```
  CLASS ClearableQueue INHERIT Queue
  OPERATION clear;
            BEGIN ..... END;
  END ClearableQueue.

-----------------------------------------------------------------

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

Figure 10: State-based guards in superclass: no inheritance anomaly

The biggest problem with synchronization expressions is that they do not blend well with inheritance. This is due to their centralized nature. It is usually impossible to inherit a synchronization expression and take new operations in the subclass into account by just making an incremental change to the synchronization behaviour, adding a new expression. Thus, redefinition of the complete synchronization expression is usually inevitable - a grave inheritance anomaly.

3.2.2 Exclusion synchronization

Until now we have tacitly assumed that an object is never invoked when one of its operations is already executing. However, activities meeting at an object are the rule rather than the exception. So the object designer has to specify the object's behaviour under such circumstances. The sequential specification of the object has to be extended to cope with a concurrent environment.

The simplest extension is to postulate *serializability*: the effect of any concurrent invocation of operations and their subsequent execution is equal to the effect of *some* serial execution of the invocations (i.e., any order is acceptable). Often this is too strong a requirement. There are many cases where it is perfectly acceptable that an object, when placed in a concurrent environment, may display a behaviour unseen in a sequential environment.

In any case, the implementation usually has to employ synchronization mechanisms to meet the specification. This kind of synchronization is called *exclusion synchronization* or *concurrency control*. Note that how to synchronize is highly dependent on the object's representation. Complete mutual exclusion as known from monitors guarantees serializability. So do transactional locking schemes such as two-phase locking. But even more liberal synchronization measures are often sufficient when the specification does not require serializability.

The integrative approach calls for avoidance of explicit locking operations: the programming language should offer transactions or declarative constructs suitable for specifying exclusion. Several languages are rather poor in this respect, distinguishing just between *atomic objects* with complete exclusion (i.e., monitor-like) and non-atomic objects with no exclusion at all. Some insist that all objects be atomic

- an approach that is doomed to failure, for the same reasons that make nested monitors impractical.

Non-atomic shared objects give rise to *intra-object concurrency*, in addition to the *inter-object concurrency* which is present a priori, due to the mere existence of concurrent activities operating on different objects. We distinguish between different degrees of intra-object concurrency:

1. *Atomic object:* There is no intra-object concurrency.
2. *Quasi-atomic object:* Several activations of operations may coexist, but at most one of them is not suspended. (This is similar to a monitor using event variables to suspend processes.)
3. *Semi-concurrent object:* There is true intra-object concurrency, but some restrictions apply, as specified by the programmer.
4. *Fully concurrent object:* Concurrency within the object is not restricted. This is the default for a normal sequential object placed in a concurrent environment.

We will often not distinguish between semi-concurrent and fully concurrent objects, just calling them concurrent objects.

Exclusion can be specified either per object/class or per operation. The per-object approach lacks flexibility and results in declaring objects either fully concurrent or atomic. While this may be considered restrictive, even more restrictive schemes have been suggested: it has been argued that intra-object concurrency should be banished altogether because reasoning about programs that contain atomic objects only is much easier [America 89] [Meyer 97].

Most language designers have considered this too severe a restriction. It is common to allow intra-object concurrency and to tie the specification of exclusion to the operations. A typical device is the `synchronized` keyword in *Java*: operations marked `synchronized` are mutually exclusive for the underlying object (or for the underlying class if the operation is a `static` method). Actually, `synchronized` can also be used for establishing arbitrary critical regions. Condition synchronization is handled using events, another indication that Java favours a low degree of integration.

A language that allows for specifying reader/writer exclusion is *Distributed Eiffel* [Gunaseelan/LeBlanc 92], designed as a modified Eiffel for programming distributed applications on top of the Clouds distributed operating system. An operation can be marked as `accesses` or `modifies`, meaning that it has to acquire a read lock or a write lock, resp., on the object before it can execute. If neither mark is present, no lock is acquired.

This approach is generalized in another Eiffel extension, *CEiffel* [Löhr 93]: using *annotations* to the operations, a binary, symmetric *compatibility* relation among the operations of an object can be specified. If operation `op1` is declared compatible to operation `op2`, both can be executed in an overlapping fashion. Incompatible operations are mutually exclusive. This approach can be traced back to [Andrews/McGraw 77] where a centralized `parallel` clause is used to specify compatibility in a precursor of SR. 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 atomic 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.

The fact that exclusion synchronization is implementation-dependent and therefore conceptually different from condition synchronization is recognized by some, though not all, languages. *Guide* employs counters not only for condition synchronization, as mentioned above, but also for exclusion synchronization. `current(op)` denotes the number of activities currently executing `op`, so a guard `op1:   current(op2)=0` specifies that `op2` excludes `op1` (though not the other way around). Unfortunately, this approach is once more prone to inheritance anomalies. Figure 11 shows three versions of a class `Part`, given in Distributed Eiffel, CEiffel and Guide, resp. A *CEiffel annotation* starts with the characters `--`, just like an Eiffel comment, but is identified as an annotation by the next character. (The reader is invited to fill in the missing code.)

```
    CLASS Part
    EXPORT number, text, update
    FEATURE number: Integer IS
            DO ..... END;

            text: String ACCESSES
            DO ..... END;

            update(s: String) MODIFIES
            DO ..... END;

            .....
    END -- Part
  -----------------------------------------------------------

    CLASS Part
    CREATION ...
    FEATURE number: Integer IS   --|| number --
            DO ..... END:

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

            update(s: String) IS --|| number --
            DO ..... END;

            .....
    END -- Part
```

```
 ---------------------------------------------------------------

   CLASS Part IMPLEMENTS PartType IS
         .....

   CONTROL text:   current(update) = 0;
           update: current(update) + current(text) = 0;
   END Part.
```

Figure 11: Reader/Writer exclusion using Distributed Eiffel, CEiffel and Guide

A subclass `ClearablePart` is shown in Figure 12. It turns out that the Guide version suffers from an inheritance anomaly: although `text` and `update` are not redefined, their guards have to be [2].

```
CLASS ClearablePart INHERIT Part
EXPORT number, text, update, clear
FEATURE clear MODIFIES
        DO ..... END;
END -- ClearablePart
--------------------------------------------------------------------

CLASS ClearablePart INHERIT Part
CREATION ...
FEATURE clear IS --|| number --
        DO ..... END;
END -- ClearablePart
--------------------------------------------------------------------

CLASS ClearablePart SUBCLASS OF Part IMPLEMENTS ClearablePartType IS
METHOD clear;
       BEGIN ..... END clear;
CONTROL text:   current(update) + current(clear) = 0;
        update: current(update) + current(clear) + current(text) = 0;
        clear:  current(update) + current(clear) + current(text) = 0;
END ClearablePart.
```

Figure 12: Exclusion synchronization and inheritance
in Distributed Eiffel, CEiffel and Guide

A unique approach to concurrency control for shared passive objects is taken in the parallel language *Charm++* [Kalé/Krishnan 93]. A few special, built-in, templates for abstract classes support concepts such as "accumulator objects", "monotonic objects" and others. These concepts are defined by certain properties of the

[2]It should be mentioned that the current version of Guide supports the extension - by logical *and*ing - of an inherited guard. So if we *and* assertions `current(clear)=0` instead of summing up the synchronization counters we can at least weaken the anomaly in this example.

operations. A *monotonic class* MC, e.g., models objects that are modified monotonically (with respect to some linear ordering) through an update operation. By virtue of being declared monotonic, MC has a built-in operation MonoValue() that delivers the current value of the object. The programmer has to explicitly provide the operation update that has to be monotonic, idempotent, commutative and associative. An example is given in Figure 13: a Max object m would contain some current maximum integer value which is updated periodically, using m->update(new). The current maximum value is obtained by m->MonoValue(). The necessary synchronization is performed automatically (and so is the replica maintenance of its distributed implementation, as explained below). The programmer only supplies the problem-specific data representation, which must be a Charm++ message object (an instance of a special record-like message type), and the update code.

```
message Integer {int value};

monotonic class Max {
        Integer *object;
public: Max(Integer *init) {
              object = (Integer *) new_message(Integer);
              object->value = init->value; }

        int update(Integer *new) {
              if (object->value < new->value) {
                  object->value = new->value;
                  return(1); }
              return(0); }
};
```

Figure 13: Simple monotonic class in Charm++

Providing built-in solutions for several common synchronization problems represents, in a way, the ultimate integration of synchronization. The advantages are obvious - and so are the drawbacks: if you need support for something the language designers did not plan for, you may find yourself building a simple mechanism out of heavy-weight constructs.

3.2.3 Scheduling

There are two reasons why an object might not accept an invocation right when it arrives: condition synchronization and exclusion synchronization. An invocation that has been issued yet not accepted is said to be *pending.* Any synchronized object has an associated queue of pending invocations.

The natural strategy for handling this queue is FCFS: if several pending invocations become eligible for acceptance, they are accepted in the sequence of their arrival. This strategy is not necessarily fair (as seen, e.g., with the Reader/Writer problem), and it may not be in accordance with what the programmer wants. The situation can be handled in either of two ways: a) the language designer ignores it, leaving the programmer with the task of designing separate scheduling objects which implement the desired non-standard strategy; b) hooks are built into the language that allow to refer to the pending invocations in appropriate ways.

Most languages support b), if only in restricted form. A popular approach is to use synchronization counters which, as we have seen, are also used for condition and exclusion synchronization. Simple scheduling problems (such as some Reader/Writer variants) can be solved in this way; but synchronization counters are much too restrictive, not allowing to take into account object state, operation parameters and invocation time. Figure 14 shows another solution to the Reader/Writer problem solved in Figure 11; this time, priority is given to the writers by extending the `text` guard with `pending(update)=0`. Note that exclusion issues and scheduling issues are indiscriminately interwoven, which may hamper understanding and complicates modification in subclasses.

```
CLASS Part IMPLEMENTS PartType IS
        .....
CONTROL text:   current(update) + pending(update) = 0;
        update: current(update) + current(text)   = 0;
END Part.
```

Figure 14: Using synchronization counters
for both exclusion and scheduling in Guide

The language SR, although not object-oriented, offers a more powerful construct for scheduling: invocations of an operation declared with a `by` clause are scheduled according to the value of the integer expression given in that clause (which may refer to parameters). A flexible solution that is even more powerful than SR's `by` has been described in [McHale et al. 91] and [Löhr 91]: *scheduling predicates* referring to the pending invocations are used in guards. Their expressivenes is strong enough to allow for straightforward solutions of complex scheduling problems. Figure 15 shows the `Part` example (writer priority) and an atomic, shortest-job-next `Printer` class.

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

        text: String IS      --|| text, number --
        REQUIRE --@--
                ALL update SAT false  -- no pending updates
        DO ..... END;

        update(s: String) IS --|| number --
        DO ..... END;
END -- Part
```

```
----------------------------------------------------------------

CLASS Printer
      .....
FEATURE print(f: File) IS
        REQUIRE --@--
                ALL print SAT
                  ( print.f.size>=f.size AND
                    print.f.size =f.size => print.Rank>=Rank )
        DO ..... END
END -- Printer
```

Figure 15: Scheduling predicates (CEiffel version)

The examples are given in CEiffel where a guard is marked by the *delay annotation* `--@--` in Eiffel's precondition clause (`REQUIRE ...`). The universal quantifier in the guard of the operation `print` uses a variable called `print`, too. This implicitly declared variable refers to the pending `print` requests, represented as records of invocation arguments. `Rank` is an additional record field referring to the virtual arrival time of an invocation. Here it is used for FCFS ordering of jobs with equal size.

3.2.4 Access to object representation considered harmful

There is a potential for subtle errors when synchronization and scheduling decisions for an object are based on those very variables that represent the object's state and are modified by the object's operations. The programmer has to be aware of possible conflicts between guard evaluation and state manipulation. Enhanced safety is achieved by complete separation of instance variables and *synchronization variables* [McHale 94]. As the `Printer` example in Figure 16 demonstrates, all synchronization information is centralized in a `synchronization` section of the class. Synchronization variables, actions and guards are declared here. Several standard identifiers are used to refer to action-triggering events such as `arrival` of an invocation, `start` of an operation etc. Other identifiers refer to synchronization counters (such as `exec`, `waiting` etc.) and to certain constituents of a guard (e.g., `this_inv`).

The first line of the synchronization section declares an integer variable `len` of which there will be one instance per invocation of `print`. The second line specifies an action to be taken when a `print` invocation arrives: get the length of the file to be printed and store it in this invocation's `len` variable. The remaining four lines constitute the guard for `print`, specifying mutual exclusion in the first conjunct and shortest-job-next scheduling in the second conjunct.

The synchronization variables approach lends itself naturally to the introduction of generic *synchronization policies* which can be instantiated for different application classes that need the same kind of synchronization. This can also mitigate the problem of inheritance anomalies in some, though not all, cases.

```
class Printer {
      print(String fileName) {.....}

synchronization
      int len local to print;
      arrival(print) -> this_inv.len := ...; // get file length
      print: exec(print)=0 and
             there_is_no(p in waiting(print):
               p.len<this_inv.len or
               p.len=this_inv.len and p.arr_time<this_inv.arr_time);
}
```

Figure 16: Synchronization variables
(example shows one variable, one action and one guard)

3.3 Active objects

Invocation of a *passive object* works like a procedure call: the calling activity "enters" an operation of the object, executes the operation and returns. An *active object*, however, can have one or more independent threads of control associated with it. Now introducing low-level message-passing primitives for inter-object communication would violate the spirit of integration. It is preferable to keep the invocation paradigm and generalize it towards *remote invocation* as known from Ada. It is important to differentiate between this notion of remote invocation - meaning *active object invocation* - and the RPC-like notion of remote invocation, introduced in 2.3.2 as a distributed implementation technique for *passive object invocation* across machine boundaries.

3.3.1 Object bodies

Many designs for the active object concept have been inspired by the Ada tasking model. A task has its own thread of control. Its *body* 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. As opposed to a passive object, however, a task uses *explicit accept* statements for accepting invocations and executing the requested service. Compare this to the *implicit acceptance* in a passive object: if a body is present, it is used for initialization only; and acceptance just means that one of the operations starts executing.

Concurrent C++ and Pool[3] are typical representatives of this approach. We omit Concurrent C++ because of its low level of integration (3.1.2). A Pool class for active `Queue` objects is shown in Figure 17. The declarative part for local data is omitted. Operations such as `METHOD enq` are declared just like for passive objects. A `Queue` object has a single thread of control. Its activity is described by the statements enclosed in the `BODY/YDOB` keywords (`DO/OD` is an infinite loop). As opposed to Ada,

[3]Actually, there are three different versions of Pool: Pool-T, Pool2 and Pool-I

the accept statements, starting with the keyword `ANSWER`, just refer to one or more operation names (`ANY` meaning all operation names).

```
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);
              %% postprocessing starts:
       front := (front+1)MOD size END deq

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

Figure 17: Active object class in Pool
(comments start with `%%` and end with line end)

Several points have to be noted:

1. A queue is usually implemented as a passive object, as we have seen earlier. 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.
2. The example does exhibit tiny bits of independent activity:
   - The caller of an operation continues as soon as the operation's `RESULT` statement has been executed. Any statements following the `RESULT` are executed by the object, which constitutes a kind of independent *postprocessing.* The service is therefore *partly asynchronous.*
   - The invocation of an operation without a result (`enq` in the example) returns immediately. In this case, the service is *fully asynchronous.*
3. Notice that a missing body defaults to

   ```
   DO ANSWER ANY OD
   ```

   This is Pool's answer to the question of how to distinguish passive from active objects: there is no such distinction. All objects are conceptually active. Whether an object without a body is implemented with or without a permanent thread is a matter of optimization (in a centralized system) and can be left to the compiler. Note, however, that because of the single-thread semantics those "passive" objects are atomic (monitor-like), not concurrent.

An important observation is that in more complex cases the body may describe both application-specific behaviour and the logic for accepting invocations, i.e., all condition synchronization and scheduling (not exclusion synchronization, because of the atomicity) [4]. The missing distinction among these very different issues and their centralized handling in the body is the source of several problems with Pool. The language is highly prone to inheritance anomalies. If Guide's central control clause was problematic, Pool's `BODY` is even more so because it is imperative rather than declarative, lacking the structure achieved by associating guards with operations. Figure 18 shows the Pool version of the `ClearableQueue` we have seen before. 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.

```
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 18: Pool: even a very simple subclass necessitates a new body

3.3.2 Asynchronous service and lazy synchronization

We have seen that Pool supports limited forms of asynchronous service: if there is no result, the client continues immediately after the call (not even waiting for the acceptance of the call); and if there is a result, client and server operate concurrently after the result has been returned (postprocessing).

It is possible to decouple invocation and waiting for a result. Using *futures* as surrogates for results that are yet to be computed, asynchrony can be achieved without resort to explicit message passing. Only when the caller really needs the result - i.e., is going to operate on it - synchronization with the service provider is 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, or *lazy synchronization.* This technique was first introduced in Eiffel// [Caromel 93] where it is known as *wait-by-necessity.* 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

[4]This is not the whole story because `ANSWER` statements can also occur in the code of operations, giving rise to quasi-atomic objects.

an active object to control the acceptance of invocations in its `Live` routine, much like it is done with `ANSWER` in Pool.

Eiffel// shares with Pool the property that there are no concurrent objects. Passive objects do exist in Eiffel//: any non-`PROCESS` object is passive. But these objects cannot be shared among active objects. Objects are always passed by value in remote invocations, i.e., the active object receives a copy.

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

Figure 19: Lazy synchronization

Service execution is *always* asynchronous in Eiffel//. If there is a result, lazy synchronization takes effect. In Figure 19, the calling client may proceed immediately after the invocation, becoming blocked only when it tries to prematurely invoke the result object.

3.3.3 Implicit acceptance

We have seen that controlling the behaviour of an active object by a body is fraught with problems. Fortunately, active objects can also use implicit acceptance as known from passive objects. (We not in passing that SR is a language where both implicit and explicit acceptance can be used at the programmer's discretion.)

Exclusive usage of implicit acceptance brings forth a restricted form of active objects - *reactive objects*: they start operating only when invoked. Reactive objects degenerate to passive objects if all operations are synchronous.

Reactive objects with asynchronous operations only are supported by Charm++: they are instances of special classes that start with the keywords `chare class`. Figure 20 shows an example - a chare class featuring two operations.

```
message Message{.....};

chare class Printer {
      .....

entry: printwhite(Message *msg)
       {.....}

       printgrey (Message *msg)
       {.....}
}
```

Figure 20: A Charm++ class for reactive `Printer` objects

The operations of a chare class must have exactly one argument which has to be a pointer to a Charm++ message. Note that although the `Printer` class "looks passive" a `Printer` object is active and operates concurrently with its clients (if only executing one operation at a time). Asynchronous invocation without result is semantically equivalent to asynchronous message passing - and is in fact implemented with messages in Charm++. That the semantics is independent of explicit vs. implicit sending/receiving is the essence of the well-known *duality principle* of procedure-oriented vs. message-oriented interaction [Lauer/Needham 78].

Chare classes cannot be used with condition synchronization, nor do they allow for synchronous operations. The reason is efficient distributed implementation (see 3.4 below). There is no conceptual reason, however, why condition synchronization and discretionary synchrony/asynchrony specification should not go together. In fact, CEiffel allows a class `Printer` to be written as shown in Figure 21. Operations are synchronous by default, as for passive objects, but can be marked asynchronous, using the *asynchrony annotation* `--v--`. Guards are introduced by the *delay annotation* `--@--`. Note that `foo` is a guarded, asynchronous operation with lazy synchronization.

```
CLASS Printer
      .....
FEATURE print(f: File) IS --v--
        DO ..... END

        waitTrayEmpty  IS
        REQUIRE --@--
                sheets = 0
        DO END

        foo: T IS --v--
        REQUIRE   --@--
                .....
        DO ..... END
END -- Printer
```

Figure 21: A CEiffel class for reactive `Printer` objects

3.3.4 Multi-threaded objects

We have only considered *atomic* active objects by now: each object had a single thread of control and so was not capable of intra-object concurrency. Remembering the duality principle, we see that dynamic invocation hierarchies of single-threaded active objects are prone to the same pitfalls as nested monitors (although the problem is mitigated when using asynchronous invocation).

The more flexible *concurrent* active objects can be traced back to the *resources* of SR. In addition to data and procedures, an SR *resource* may encapsulate one or more processes. Concurrency within a resource may result both from concurrent procedure invocations and from the activities of the resource's processes.

In an object-oriented setting, multiple static processes in an object would amount to multiple bodies using explicit acceptance. Now, in view of inheritance anomalies, if one such body causes problems, multiple bodies would lead to disaster. It is therefore the cleanest solution to use only implicit acceptance (i.e., procedures).

Removing the atomicity restriction from reactive objects readily allows for concurrent reactive objects. Concurrency control can be introduced in the same way as for passive objects. Act++, another extension of C++, is an example of this approach [Kafura et al. 93]. Inspired by the actor model, Act++ supports reactive objects as instances of subclasses of a given class `Actor`. Synchronization is achieved by having an object switch between different *behaviours*, each tied to a set of operations whose invocations are acceptable by that behaviour.

Even *autonomous* - not just reactive - active objects are possible without a body. In CEiffel, operations can be specified as autonomous using the *autonomy annotation* `-->--`. The language thus allows to decide for each operation of a class whether it should be synchronous (no annotation) or asynchronous (annotation `--v--`) or autonomous (annotation `-->--`). 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 intra-object concurrency is still controlled by the compatibility annotations.

The annotations of CEiffel blend well with inheritance and avoid anomalies because any centralization, such as represented by a *body* or a central *synchronization expression*, is avoided: condition synchronization, exclusion synchronization and autonomy/asynchrony are specified *per operation* and *orthogonally* to each other. Subclassing, even with multiple inheritance, works just like in the sequential case, no matter if asynchronous or autonomous operations are involved. (The situation can be seen as a generalization of what has been described as "process inheritance" in [Thomsen 87].) The difference between a passive and an active class is just that the former has only synchronous operations.

Figure 22 shows an example with double inheritance. Class `Alien` inherits both from `Moving` (which models objects moving autonomously in the plane) and from `Beeping` (which models objects repeatedly producing a sound). Thus, `Alien` objects are autonomous and will both move and beep. Their behavior may of course be influenced by any additional operations provided in `Alien`.

```
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


CLASS Beeping CREATION init
FEATURE on(b: Boolean) IS
        DO beepon := b END;

FEATURE {}
        beepon: Boolean;
        sound: Speaker;

        beep IS -->--
        DO IF beepon THEN sound.beep END END;

        init(s: Speaker) IS
        DO sound := s END
END -- Beeping


CLASS Alien INHERIT Moving  RENAME init AS minit END;
                    Beeping RENAME init AS binit END;
CREATION init
FEATURE .......
END -- Alien
```

Figure 22: Multiple inheritance with multi-threaded active classes in CEiffel

3.4 Distribution

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

Distributed implementations are available for many concurrent object-oriented languages. Some have even been designed with distribution in mind right from the beginning (e.g., Pool, Eiffel//, Charm++). Distribution transparency sometimes suffers from this approach. Ideally, the issues of concurrent semantics on the one hand and distributed implementation on the other hand should be kept independent of each other. This would imply access transparency for all kinds of objects, active or passive, whether local or remote.

3.4.1 Accessibility and fault tolerance

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

3.4.2 Migration

In order to improve the accessibility of objects, some languages or systems support mechanisms for object migration. Object migration has been pioneered by *Emerald* [Jul et al. 88]. A more recent, and truly object-oriented, system is *Dowl* [Achauer 93], a distributed extension of the Trellis/Owl language [Moss/Kohler 87]. Dowl features a standard attribute `$location` of type `$Node` for each object. This attribute can be read and written; changing its value causes the object to *migrate* to a new node. This can even happen on the fly, i.e., while an operation is in execution.

If any attribute `a` of a class `C` is declared `a: $attached T`, with some type `T`, this implies *co-location* of any `C` object `c` and the object referred to by `c.a`. Attachment is transitive (but not symmetric).

Parameter migration towards an invoked object can be specified as either permanent (”call-by-move”) or temporary (”call-by-visit”). All these features have been adapted from similar features of Emerald.

Explicit migration control is of course not distribution-transparent. It can also be argued that the programmer may not have sufficient knowledge to use those language features in an efficient way. Implicit migration control, combined with clever heuristics, may be an alternative (see 4.5.2 below).

3.4.3 Replication

As for migration, a first motivation of replication is in increasing the accessibility of an object, by replicating it onto the processors of its (remote) clients. A second motivation is fault-tolerance. By replicating an object on several processors, its services become robust against possible processor failure. In both cases, a fundamental issue is to maintain the consistency of the replicas, i.e. to ensure that all replicas hold the same values. In the Electra [Maffeis 95] distributed system, the concept of remote invocation has been extended in the following fashion: invoking an object leads to the invocation of all its replicas while ensuring that concurrent invocations are ordered along the same (total) order for all replicas. A system supporting weaker orderings, exploiting knowledge about the semantics of the operations, is described in [Huang 94]. A general mechanism for group invocation that is well-suited for replicated objects was introduced in [Black/Immel 93].

3.5 Parallelism

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.

We have already seen EPEE, an SPMD *library* for Eiffel (2.2). Charm++, also mentioned above, is an example of the *integrative* approach. We have not mentioned Charm's specific abilities for SPMD yet. There exists a variant of the chare class concept called the *branched chare class*, instances of which are called branched chares. A branched chare is a distributed object: its code is replicated among the nodes of a distributed-memory computer (or of a computer network) and each node works on one fragment of the object.

Similar as they may seem, there is a big difference between the object models of EPEE and Charm++: the interface of a branched chare class reflects the fragmentation in that it describes the messages (or, dually, the asynchronous invocations) it can accept *from other fragments*, in addition to messages from other objects (chares or branched chares). Thus, although EPEE has the advantage of hiding the explicit operations for inter-fragment message passing from the clients of an object, programming in Charm++ is less cumbersome because message passing is built into the language - as chare invocation.

The ubiquity of C++ has given rise to a variety of approaches to parallel programming based on that language [Wilson/Lu 96], most of them integrative. The majority tries to avoid explicit message passing so that the object model is not blurred by a different - and lower-level - paradigm.

3.6 Limitations of the integrative approach

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

3.6.1 Inheritance anomaly

Inheritance 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, as we have seen in several examples, (1) synchronization is difficult to specify and even more difficult to reuse, because of the high interdependency between the synchronization conditions for different methods, (2) various uses of inheritance (inheriting variables, methods, synchronizations) may conflict with each other, as noted in [McHale 94] [Baquero et al. 95]. In some cases, defining a new subclass, even only with one additional method, may force the redefinition of all synchronization specifications.

Specifications along centralized schemes turn out to be very difficult to reuse, and often must be completely redefined. Decentralized schemes, being modular by essence, are better suited for selective specialization. However, this fine-grained decomposition, down to the level of each method, may also fail to solve the problem. This is because synchronization specifications, even if decomposed for each method, may still remain interdependent: we have seen in 3.2.1 that for intra-object synchronization with synchronization counters, adding a `delete` method in a subclass forces the redefinition of the `remove` guard. (See [Matsuoka/Yonezawa 93] for a detailed analysis and classification of the possible problems.)

Recent directions proposed for minimizing the problem include the following: (1) specifying and specializing independently condition and exclusion synchronization [Thomas 92] and autonomy/asynchrony [Löhr 93], (2) shunning synchronization counters, (3) allowing the programmer to select among several schemes [Matsuoka/Yonezawa 93] and (4) generic synchronization policies (3.2.4) as an alternative to inheritance for reusing synchronization specifications [McHale 94].

3.6.2 Compatibility of transaction protocols

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

3.6.3 Replication of objects and communications

The communication protocols which have been designed for fault-tolerant distributed computing (see Sect. 3.4.3) consider a standard client/server model. The straightforward transfer of such protocols to the object model leads to the problem of unexpected duplication of invocations. An object often acts both as a client and as a server. Thus an object which has been replicated as a server may in turn invoke other objects (as a client). As a result, all replicas of the object will invoke these other objects several times. This unexpected duplication of invocations may lead, in the best case, to inefficiency, in the worst case to inconsistencies. A solution, proposed in [Mazouni et al. 95], is based on *pre-filtering* and *post-filtering*. Pre-filtering consists of coordinating processing by the replicas (when considered as a client) in order to generate a single invocation. Post-filtering is the dual operation for the replicas (when considered as a server) in order to discard redundant invocations.

3.6.4 Factorization vs. distribution

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

The concept of *class variables*, supported by several object-oriented programming languages, is difficult and expensive to implement for a distributed system. Unless introducing complex and costly transaction mechanisms, consistency is hard to maintain once instances of a same class are distributed among processors. Note that this is a general problem 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 [Wolff 95] 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 with partitioning the class code among processors.

A rather different approach replaces the inheritance mechanism between classes with the concept/mechanism of *delegation* between objects. This mechanism has actually been introduced in the actor concurrent programming language Act 1 [Lieberman 87]. Intuitively, an object which does not understand a message will then delegate it (i.e. forward it[5]) to another object, called its *proxy*. The proxy will process the message in place of the initial receiver, or it can also itself delegate it further to its own designated proxy. This alternative to inheritance is very appealing as it only relies on message passing; thus it blends well with distributed implementation. Note, however, that the delegation mechanism needs some non-trivial synchronization mechanism to ensure the proper handling (ordering) of recursive messages, prior to other incoming messages. Thus, it may not offer a general and complete alternative solution [Briot/Yonezawa 87].

3.7 Evaluation of the integrative approach

In summary, the integrative approach is appealing because it merges concepts from object-oriented and distributed programming. It thus presents a minimal number of concepts and a single conceptual framework to the programmer. Nevertheless,

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

as we discussed in Sect. 3.6, this approach unfortunately suffers from limitations in some aspects of the integration.

Another potential weakness is that a too systematic unification/integration may lead to a too restrictive model ("too much uniformity kills variety !") and to inefficiencies. For instance, stating that every object is active (or that every message transmission is a transaction) may be inappropriate for some applications not necessarily requiring such protocols and their associated computational load. Last but not least, we may have a *legacy problem*: difficulties with reusing standard sequential programs. Encapsulation of sequential programs into active objects may look like a straightforward solution. But note that cohabitation of active and passive objects may require the observance of certain methodological rules [Caromel 93].

4 The Reflective approach

As we discussed earlier, the library approach helps with structuring concurrent programming concepts and mechanisms, due to encapsulation, genericity, class, and inheritance concepts. The integrative approach minimizes the amount of concepts to be mastered by the programmer and makes mechanisms more transparent, but at the cost of possibly reducing the flexibility and the efficiency of the mechanisms offered. Indeed programming systems built from libraries are often more extensible than languages designed along the integrative approach. Libraries help at structuring and simulating various solutions, and thus usually bring good flexibility, whereas brand new languages may freeze their computation and communication models too early. In other words it would be interesting to keep the unification and simplification advantages of the integrative approach, while retaining the flexibility of the applicative/library approach.

One important observation is that the 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. In other words, the end user programs his or her 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, following the library approach.

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

4.1 Reflection

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

Reflection helps at decorrelating libraries specifying implementation and execution models (execution strategies, concurrency control, object distribution) from the application program. This increases modularity, readability and reusability of programs. Last, reflection provides a methodology to open up and make adaptable, through a *meta-interface*[6], implementation decisions and resources management, which are often hard-wired and fixed, or delegated by the programming language to the underlying operating system.

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

4.2 Reflection and objects

Reflection fits especially well with object concepts, which enforce a good encapsulation of levels and a modularity of effects. It is therefore natural to organize the control of the behaviour 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. 91], and its components are called *meta-objects* [Maes 87], as meta-programs are represented by objects. They may represent various characteristics of the execution context such as: representation, implementation, execution, communication and location. By specializing meta-objects we may extend and modify, locally, the execution context of some specific objects of the application program.

Reflection may also help with expressing and controlling resources management, not only at the level of an individual object, but also at a broader level such as scheduler, processor, name space, object group..., such resources also being represented by meta-objects. This allows for a very fine-grained control (e.g. for scheduling and load balancing) with the whole expressive power of a full programming language [Okamura/Ishikawa 94], as opposed to some global and fixed algorithm (which is usually optimized for a specific kind of application or an average case).

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

4.3 Examples of meta-object protocols (MOPs)

The CodA architecture [McAffer 95] is a representative example of a general object-based reflective architecture (i.e. a "MOP") based on *meta-components*[7]. CodA considers by default seven (7) meta-components, associated to each object, corresponding to: *message sending*, *receiving*, *buffering*, *selection*, *method lookup*, *execution*, and *state accessing*. An object with default meta-components behaves like a standard (sequential and passive) object[8]. Attaching specific (specialized) meta-components allows to selectively changing a specific aspect of the representation or execution model for a single object. A standard interface between meta-components supports composing meta-components from different origins.

Note that some other reflective architectures may be more specialized and may offer a more reduced (and abstract) set of meta-components. Examples are the Actalk and GARF platforms, where a smaller amount of meta-components may be in practice sufficient to express a large variety of schemes and application problems. The Actalk platform [Briot 89] [Briot 96] helps at experimenting with various synchronization and communication models for a given program, by changing and specializing various models/components of: (1) *activity* (implicit or explicit acceptance of requests, intra-objet concurrency etc.) and *synchronization* (abstract behaviours, guards etc.), (2) *communication* (synchronous, asynchronous etc.), and (3) *invocation* (time stamp, priority etc.). The GARF platform [Garbinato et al. 94], 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.).

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

Smalltalk is a representative example of that latter category. In addition to the (meta-)representation of the program structures and mechanisms, as first class objects (see Sect. 2.1), a few very powerful reflective mechanisms offer some control over program execution. Examples are: redefinition of error handling message, reference to current context, references swap, changing the class of an object etc. Such mechanisms facilitate building and integrating various platforms for concurrent, parallel and distributed programming, such as Simtalk, Actalk, GARF, and CodA itself.

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

[8]To be more precise, as a standard Smalltalk object, as CodA is currently implemented in Smalltalk.

4.4 Examples of applications

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

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

4.4.1 Concurrency models

In order to introduce concurrency for a given object (by making it into an *active* object, following the integrated approach), two meta-components are specialized. The specialized *message buffering* component[9] is a queue which will buffer incoming messages. The specialized *execution* component associates an independent activity (thread) with the object. This thread executes an infinite loop that accepts messages from the *buffering* component.

4.4.2 Distribution models

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

4.4.3 Migration and replication models

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

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

4.5 Other Examples of reflective architectures

We will briefly mention other related examples of representative reflective architectures and their applications, not trying to be exhaustive.

4.5.1 Dynamic installation and composition of protocols

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

4.5.2 Control of migration

The autonomy and self-containedness of objects, further reinforced in the case of active objects, makes them easier to migrate "as a single piece". Nevertheless, the decision to migrate an object is an important issue which often remains with the programmer. As mentioned in 3.4.2, it may be interesting to semi-automate such a decision, along various considerations such as processor load, ratio of remote communications etc. Reflection helps with integrating such statistical data (residing for physical and shared resources) and with using them by various migration algorithms described at the meta-level [Okamura/Ishikawa 94].

4.5.3 Customizing system policies

The Apertos distributed operating system [Yokote 92] represents a significant and innovative example of a distributed operating system, completely designed with an object-based reflective architecture (MOP). In addition to the modularity and the genericity of the architecture of systems like Choices or Peace, reflection opens up another dimension of (possibly dynamic) customization of the system towards application requirements. For example, we can easily specialize the scheduling policy in order to support various kinds of schedulers, e.g. a real-time scheduler. Another gain is in the size of the micro-kernel obtained, which is particularly small, as it is reduced to supporting the basic reflective operations and the basic resources abstractions. This facilitates both the understanding and the porting of the system.

4.5.4 Reflective extension of an existing commercial system

A reflective methodology has recently been used in order to incorporate extended[10] 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 introduce features such as lock delegation, dependency tracking between transactions and definition of conflicts, and to represent them

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

as reflective operations [Barga/Pu 95]. These reflective primitives are then used to implement various extended transaction models, such as split/join, cooperative groups etc.

4.6 Related techniques for customizing behavior

We finally mention two examples of customizing computational behavior that are closely related to reflection.

4.6.1 The composition-filters model

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

4.6.2 Generic run-time as a dual approach

The boundary between programming languages and operating systems is getting thinner. Reflective programming languages have some high-level representation of the underlying execution model. Conversely, and dual to reflection, several distributed operating systems provide a generic run time layer, as for instance the COOL layer for the Chorus operating system [Lea et al. 93]. These generic run time layers are designed to be used by various programming languages: "upcalls" are used to delegate specific representation decisions to the programming language.

4.7 Evaluation of the reflective approach

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

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

5 Conclusion

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

The *library* approach helps with structuring concurrent programming concepts and mechanisms through encapsulation, genericity, classes and inheritance. Its principal limitation is that the solution of an application problem 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 amount of concepts to be mastered by the programmer and makes mechanisms more transparent, by providing a unified concurrent high-level object model. However, this is at the cost of possibly reducing the flexibility and efficiency of the mechanisms. The integrative approach can be viewed as a top-down approach and is directed towards application builders.

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

The reflective approach also contributes to blurring the distinction between programming language, operating system and data base, and at easing the development, adaptation and optimization of a minimal computing system which is dynamically extensible. Nevertheless, we should always keep in mind that this does not free us from the necessity of a good basic design and a sound set of fundamental abstractions.

References

[Achauer 93] ACHAUER, B., 1993. The Dowl distributed object-oriented language. *Comm. ACM 36(9)*, 48–55.

[Ada 83] ADA 1983. *The Programming Language Ada Reference Manual.* LNCS 155, Springer-Verlag.

[Ada 95] ADA 1995. *Ada 95 Rationale.* Intermetrics Inc, Cambridge, Mass.

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

[Agha et al. 89] AGHA, G.A., WEGNER, P., YONEZAWA, A., EDS., 1989. *Proc. ACM Sigplan Workshop on Object-Based Concurrent Programming. ACM Sigplan Not. 24(4).*

[Agha et al. 91] AGHA, G.A., HEWITT, C., WEGNER, P., YONEZAWA, A., EDS., 1991. *Proc. OOPSLA/ECOOP '90 Workshop on Object-Based Concurrent Programming. ACM OOPS Messenger 2(2).*

[Agha et al. 93a] AGHA, G.A., WEGNER, P., YONEZAWA, A., EDS., 1993. *Research Directions in Concurrent Object-Oriented Programming,* M.I.T. Press.

[Agha et al. 93b] AGHA, G.A., FRØLUND, S., PANWAR, R., STURMAN, D., 1993. A linguistic framework for dynamic composition of dependability protocols. *Dependable Computing for Critical Applications III (DCCA-3)*, IFIP Transactions, Elsevier, 197–207.

[Aksit et al. 94a] AKSIT, M., WAKITA, K., BOSCH, J., BERGMANS, L. YONEZAWA, A., 1994. Abstracting object interactions using composition filters. In [Guerraoui et al. 94], 152–184.

[Aksit et al. 94b] AKSIT, M., BOSCH, J., VAN DER STERREN, W., BERGMANS, L., 1994. Real-time specification inheritance anomalies and real-time filters. *Proc. European Conf. on Object-Oriented Programming (ECOOP '94),* LNCS 821, Springer-Verlag, 386–407.

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

[America 88] AMERICA, P.H.M., 1988. Definition of Pool2, a parallel object-oriented language. ESPRIT project 415-A, report 364. Philips Research Laboratories.

[America 89] AMERICA, P.H.M., 1989. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing 1,* 366-411

[America/van der Linden 90] AMERICA, P.H.M., VAN DER LINDEN, F., 1990. A parallel object-oriented language with inheritance and subtyping. *Proc. OOPSLA/ECOOP '90, ACM Sigplan Not. 25(10),* 161–168.

[Andersen 94] ANDERSEN, B., 1994. A general, fine-grained, machine-independent, object-oriented language. *ACM Sigplan Not. 29(5),* 17–26.

[Andler 79] ANDLER, S., 1979. Predicate Path Expressions. *Proc. 6. ACM Symp. on Principles of Programming Languages,* 226-236.

[Andrews/McGraw 77] ANDREWS, G.R., MCGRAW, J.R., 1977. Language features for process interaction. *Proc. ACM Conf. on Language Design for Reliable Software, ACM Sigplan Not. 12(3),* 114-127.

[Andrews 91] ANDREWS, G.R., 1991. *Concurrent Programming - Principles and Practice,* Benjamin/Cummings.

[Andrews/Olsson 93] ANDREWS, G.R., OLSSON, R.A., 1993. *The SR Programming Language,* Benjamin/Cummings.

[Arnold/Gosling 96] ARNOLD, K., GOSLING, J., 1996. *The Java Programming Language,* Addison-Wesley.

[Atkinson 91] ATKINSON, C., 1991. *Object-Oriented Reuse, Concurrency and Distribution,* Addison-Wesley.

[Atkinson et al. 91] ATKINSON, C., GOLDSACK, S.J., DI MAIO, A., BAYAN, R., 1991. Object-oriented concurrency and distribution in Dragoon. *J. of Object-Oriented Programming, March/April 1991,* 11–20.

[Bahsoun et al. 90] BAHSOUN, J.P., FERAUD, L., BETOURNÉ, C., 1990. The "two degrees of freedom" approach for parallel programming. *Proc. 1. Int. Conf. on Computer Languages,* IEEE, 261–270.

[Bal et al. 92] BAL, H.E., KAASHOEK, M.F., TANENBAUM, A.S., 1992. Orca: a language for parallel programming of distributed systems. *IEEE Trans. on Software Engineering 18(3),* 190–205.

[Balter et al. 94] BALTER, R., LACOURTE, S., RIVEILL, M., 1994. The Guide language. *The Computer Journal 37(6)*, 519–530.

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

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

[Barnes 97] BARNES, J., 1997. *Ada 95 Rationale: The Language, The Standard Libraries,* Springer-Verlag.

[Bergmans 94] BERGMANS, L., 1994. *Composing Concurrent Objects.* Ph.D. thesis, Universiteit Twente.

[Bernstein et al. 87] BERNSTEIN, P., HADZILACOS, V., GOODMAN, N., 1987. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley.

[Bershad et al. 88] BERSHAD, B.N., LAZOWSKA, E.D., LEVY, H.M., 1988. PRESTO: a system for object-oriented parallel programming. *Software - Practice and Experience 18(8),* 713–732.

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

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

[Black et al. 87] Black, A.P., Hutchinson, N., Jul, E., Levy, H., Carter, L., 1987. Distribution and abstract types in Emerald. *IEEE Trans. on Software Engineering 13(1),* 65–76.

[Black 91] Black, A.P., 1991. Understanding transactions in the operating system context. *Operating Systems Review 25*, 73–77.

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

[Boles 93] Boles, D., 1993. Parallel object-oriented programming with QPC++. *Structured Programming 14(4),* 158–172.

[van den Bos/Laffra 91] van den Bos, J., Laffra, C., 1991. Procol - a concurrent object-oriented language with protocols, delegation and constraints. *Acta Informatica 28,* 511–538.

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

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

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

[Briot et al. 95] Briot, J.-P., Geib, J.-M., Yonezawa, A., Eds., 1995. *Proc. France-Japan Workshop on Object-Based Parallel and Distributed Computation,* LNCS 1107, Springer-Verlag.

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

[Bruin et al. 94] de Bruin, H., Bouwman, P., van den Bos, J., 1994. Taming concurrency in Smalltalk: the Procol approach. *Object-Oriented Systems 1(1),* 45–59.

[Buhr et al. 92] Buhr, P.A., Ditchfield, G., Stroobosscher, R.A., Younger, B.M., 1992. μC++: concurrency in the object-oriented language C++. *Software - Practice and Experience 22(2),* 137–172.

[Campbell/Habermann 74] Campbell, R.H., Habermann, A.N., 1974. The specification of process synchronization by path expressions. In Gelenbe, E., Kaiser, C., Eds.: *Operating Systems,* LNCS 16, Springer-Verlag, 89–102.

[Campbell et al. 93] Campbell, R., Islam, N., Raila, D., Madany, P., 1993. Designing and implementing Choices: an object-oriented system in C++. *Comm. ACM 36(9)*, 117–126.

[Capobianchi et al. 92] Capobianchi, R., Guerraoui, R., Lanusse, A., Roux, P., 1992. Lessons from implementing active objects on a parallel machine. *Usenix Symp. on Experiences with Distributed and Multiprocessor Systems*, 13–27.

[Caromel 89] Caromel, D., 1989. Service, asynchrony and wait-by-necessity. *J. of Object-Oriented Programming 2(4),* 12–22.

[Caromel 90] Caromel, D., 1990. Concurrency and reusability: from sequential to parallel. *J. of Object-Oriented Programming 3(3),* 34–42.

[Caromel 93] Caromel, D., 1993. Towards a method of object-oriented concurrent programming. *Comm. ACM 36(9),* 90–102.

[Chandra et al. 94] Chandra, R., Gupta, A., Hennessy, J.L., 1994. COOL: an object-based language for parallel programming. *IEEE Computer 27(8),* 13–26.

[Chandy/Kesselman 93] Chandy, K.M., Kesselman, C., 1993. CC++: a declarative concurrent object-oriented programming notation. In [Agha et al. 93], 281-313.

[Chien 93a] Chien, A.A., 1993. *Concurrent Aggregates.* MIT Press.

[Chien 93b] Chien, A.A., 1993. Supporting modularity in highly-parallel programs. In [Agha et al. 93].

[Ciancarini et al. 95] Ciancarini, P., Nierstrasz, O., Yonezawa, A., Eds., 1995. *Object-Based Models and Languages for Concurrent Systems* (ECOOP '94 Workshop). LNCS 924, Springer-Verlag.

[Colin/Geib 91] Colin, J.-F., Geib, J.-M., 1991. Eiffel classes for concurrent programming. *Proc. TOOLS-4, 1991,* Prentice-Hall, 23–34.

[Corradi/Leonardi 91] Corradi, A., Leonardi, L., 1991. PO constraints as tools to synchronize active objects. *J. of Object-Oriented Programming 4(6),* 41–53.

[Decouchant et al. 91] Decouchant, D., Le Dot, P., Riveill, M., Roisin, C., Rousset de Pina, X., 1991. A synchronization mechanism for an object-oriented distributed system. *Proc. 11. Int. Conf. on Distributed Programming Systems,* IEEE, 152–159.

[Feldman et al. 93] Feldman, J.A., Lim, C.-C., Rauber, Th., 1993. The shared-memory language pSather on a distributed-memory multiprocessor. *Proc. 2. Workshop on Languages, Compilers and Runtime Environments for Distributed-Memory Multiprocessors. ACM Sigplan Not. 28(1),* 17–20.

[Finke et al. 93] Finke, S., Jahn, P., Langmack, O., Löhr, K.-P., Piens, I., Wolff, Th., 1993. Distribution and inheritance in the HERON approach to heterogeneous computing. *Proc. 13. Int. Conf. on Distributed Computing Systems,* IEEE, 399–408.

[Frølund 92] Frølund, S., 1992. Inheritance of synchronization constraints in concurrent object-oriented programming languages. *Proc. European Conf. on Object-Oriented Programming (ECOOP '92),* LNCS 615, Springer-Verlag, 185–196.

[Frølund 96] FRØLUND, S., 1996. *Coordinating Distributed Objects,* MIT Press.

[Garbinato et al. 94] GARBINATO, B., GUERRAOUI, R., MAZOUNI, K.R., 1994. Distributed programming in GARF. In [Guerraoui et al. 94], 225–239.

[Gehani/Roome 88] GEHANI, N.H., ROOME, W.D., 1988. Concurrent C++: concurrent programming with class(es). *Software - Practice & Experience 16(12),* 1157–1177.

[Goldberg/Robson 89] GOLDBERG, A., ROBSON, D., 1989. *Smalltalk-80. The Language*, Addison-Wesley.

[Grimshaw et al. 93] GRIMSHAW, A.S., STRAYER, W., NARAYAN, P., 1993. Dynamic, object-oriented parallel processing. *IEEE Parallel and Distributed Technology,* 33–48.

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

[Guerraoui et al. 94] GUERRAOUI, R., NIERSTRASZ, O., RIVEILL, M., EDS., 1994. *Proc. ECOOP '93 Workshop on Object-Based Distributed Programming*, LNCS 791, Springer-Verlag.

[Guerraoui/Schiper 95] GUERRAOUI, R., SCHIPER, A., 1995. The transaction model vs virtual synchrony model: bridging the gap. In Birman, K., Cristian, F., Mattern, F., Schiper, A., Eds.: *Distributed Systems: From Theory to Practice*, LNCS 938, Springer-Verlag, 121–132.

[Guerraoui 95] GUERRAOUI, R., 1995. Modular atomic objects. *Theory and Practice of Object Systems 1(2)*, 89–99.

[Gunaseelan/LeBlanc 92] GUNASEELAN, L., LEBLANC, R.J., 1992. Distributed Eiffel: a language for programming multi-granular distributed objects. *Proc. 4. Int. Conf. on Computer Languages,* IEEE, 331–340.

[Halstead 85] HALSTEAD, R.H., 1985. Multilisp: a language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems 7(4),* 501–538.

[Hamelin et al. 94] HAMELIN, F., JÉZÉQUEL, J.-M., PRIOL, T., 1994. A multiparadigm, object-oriented parallel environment. *Proc. 8. Int. Parallel Processing Symposium,* Cancún.

[Harbison 92] HARBISON, S.P., 1992. *Modula-3,* Prentice-Hall.

[Holmes et al. 97] HOLMES, D., NOBLE, J., POTTER, J., 1997. Aspects of synchronization. *Proc. TOOLS Pacific '97,* IEEE, 7–18.

[Huang 94] HUANG, S., 1994. Developing distributed applications by semantics-based automatic replication. *Proc. 1. Asia-Pacific Software Engineering Conf.,* Tokyo 1994, 40–49.

[Ishikawa et al. 92] ISHIKAWA, Y., TOKUDA, H., MERCER, C.W., 1992. An object-oriented real-time programming language. *IEEE Computer 25(10),* 66–73.

[Issarny 93] ISSARNY, V., 1993. An exception handling mechanism for parallel object-oriented programming. *J. of Object-Oriented Programming 6(6),* 29–40.

[Jézéquel 93a] JÉZÉQUEL, J.-M., 1993. EPEE: an Eiffel environment to program distributed-memory parallel computers. *J. of Object-Oriented Programming 6(2),* 48–54.

[Jézéquel 93b] JÉZÉQUEL, J.-M., 1993. Transparent parallelization through reuse: between a compiler and a library approach. *Proc. European Conf. on Object-Oriented Programming (ECOOP'93)*, LNCS 707, Springer-Verlag, 384–405.

[Jézéquel et al. 94] JÉZÉQUEL, J.-M., GUIDEC, F., HAMELIN, F., 1994. Parallelizing object-oriented software through the reuse of parallel components. *Object-Oriented Systems 1(2),* 149–170.

[Jul et al. 88] JUL, E., LEVY, H.M., HUTCHINSON, N.C., BLACK, A.P., 1988. Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems 6(1),* 109–133.

[Kafura/Lee 90] KAFURA, D.G., LEE, K.H., 1990. Act++: building a concurrent C++ with actors. *J. of Object-Oriented Programming 3(1).*

[Kafura et al. 93] KAFURA, D., MUKHERJI, M., LAVENDER, G., 1993. Act++: A class library for concurrent programming in C++ using actors. *J. of Object-Oriented Programming, October 1993,* 47–62.

[Kalé/Krishnan 93] KALÉ, L.V., KRISHNAN, S., 1993. Charm++: a portable concurrent object-oriented system based on C++. *Proc. ACM Conf. on Object-Oriented Systems, Languages and Applications (OOPSLA '93), ACM Sigplan Not. 28,* 91–108.

[Karaorman/Bruno 93] KARAORMAN, M., BRUNO, J., 1993. Introducing concurrency to a sequential language. *Comm. ACM 36(9)*, 103–116.

[Kiczales et al. 91] KICZALES, G., DES RIVIÈRES, J., BOBROW, D., 1991. *The Art of the Meta-Object Protocol*, MIT Press.

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

[Lauer/Needham 78] LAUER, H.C., NEEDHAM, R.M., 1978. On the duality of operating system structures. *Proc. 2. Int. Symp. on Operating Systems, IRIA Rocquencourt.* Reprinted in *ACM Operating Systems Review 13(2)*, 3–19.

[Lea et al. 93] LEA, R., JACQUEMOT, C., PILLEVESSE, E., 1993. COOL: System support for distributed programming. *Comm. ACM 36(9)*, 37–47.

[Lea 97] LEA, D., 1997. *Concurrent Programming in Java,* Addison-Wesley.

[Lehrmann Madsen et al. 93] LEHRMANN MADSEN, O., MØLLER-PEDERSEN, B., NYGAARD, K., 1993. *Object-Oriented Programming in the BETA Programming Language,* Addison-Wesley.

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

[Liskov/Sheifler 83] LISKOV, B., SHEIFLER, R., 1983. Guardians and actions: linguistic support for robust, distributed programs. *ACM Trans. on Programming Languages and Systems 5(3),* 387–404.

[Löhr 91] LÖHR, K.-P., 1991. Concurrency annotations and reusability. Report B-91-13, FB Mathematik, Freie Universität Berlin.

[Löhr 92] LÖHR, K.-P., 1992. Concurrency Annotations. *Proc. ACM Conf. on Object-Oriented Systems, Languages and Applications (OOPSLA '92), ACM Sigplan Not. 27(10),* 327–340.

[Löhr 93] LÖHR, K.-P., 1993. Concurrency annotations for reusable software. *Comm. ACM 36(9),* 81–89.

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

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

[Maffeis 95] MAFFEIS, S., 1995. *Run-Time Support for Object-Oriented Distributed Programming,* PhD dissertation, Universität Zürich.

[Malony et al. 94] MALONY, A., MOHR, B., BECKMAN, P., GANNON, D., YANG, S., BODIN, F., 1994. Performance analysis of pC++: a portable data-parallel programming system for scalable parallel computers. *Proc. 8. Int. Parallel Processing Symposium,* Cancún.

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

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

[Mazouni et al. 95] MAZOUNI, K., GARBINATO, B., GUERRAOUI, R., 1995. Building reliable client-server software using actively replicated objects. *Proc. TOOLS Europe '95*, Prentice-Hall, 37–53.

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

[McHale et al. 92] MCHALE, C., WALSH, B., BAKER, S., DONNELLY, A., 1992. Scheduling Predicates. In [Tokoro et al. 92], 177–193.

[McHale 94] MCHALE, C., 1994. *Synchronization in Concurrent, Object-Oriented Languages: Expressive Power, Genericity and Inheritance,* PhD thesis, Dept. of Computer Science, Trinity College, Dublin.

[McHugh/Cahill 93] McHugh, C., Cahill, V., 1993. Eiffel**: an implementation of Eiffel on Amadeus, a persistent, distributed applications support environment. *Proc. TOOLS Europe '93,* Prentice-Hall.

[Meseguer 93] Meseguer, J., 1993. Solving the inheritance anomaly in concurrent object-oriented programming. *Proc. European Conf. on Object-Oriented Programming (ECOOP '93),* LNCS 707, Springer-Verlag, 220–246.

[Meyer 91] Meyer, B., 1991. *Eiffel: The Language.* Prentice-Hall.

[Meyer 93] Meyer, B., 1993. Systematic concurrent object-oriented programming. *Comm. ACM 36(9)*, 56–80.

[Meyer 97] Meyer, B., 1997. *Object-Oriented Software Construction, 2. ed.,* Prentice-Hall.

[Moss/Kohler 87] Moss, J.E.B., Kohler, W.H., 1987. Concurrency features for the Trellis/Owl language. *Proc. European Conf. on Object-Oriented Programming (ECOOP '87),* LNCS 276, Springer-Verlag, 171–180.

[Mowbray/Zahavi 95] Mowbray, T.J., and Zahavi, R., 1995 *The Essential CORBA: System Integration Using Distributed Objects,* John Wiley & Sons and The Object Management Group.

[Neusius 91] Neusius, C., 1991. Synchronizing actions. *Proc. European Conf. on Object-Oriented Programming (ECOOP '91),* LNCS 512, Springer-Verlag, 118–132.

[Nicol et al. 93] Nicol, J., Wilkes, T., Manola, F., 1993. Object-orientation in heterogeneous distributed computing systems. *IEEE Computer 26(6)*, 57–67.

[Nierstrasz 87] Nierstrasz, O.M., 1987. Active objects in Hybrid. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87), ACM Sigplan Not. 22(12),* 243–253.

[Nierstrasz 93a] Nierstrasz, O.M., 1993. Composing active objects. In [Agha et al. 93].

[Nierstrasz 93b] Nierstrasz, O.M., 1993. Regular types for active objects. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93), ACM Sigplan Not. 28,* 1–15.

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

[OMG 95] OMG 1995. *The Common Object Request Broker: Architecture and Specification (Revision 2.0).* Object Management Group, Framingham, Mass.

[OSF 94] OSF 1994. *DCE Application Development Guide (Revision 1.0.2).* Open Software Foundation, Cambridge, Mass.

[Papathomas 89] Papathomas, M., 1989. Concurrency issues in object-oriented programming languages. In Tsichritzis, D.C., ed.: *Object-Oriented Development,* Centre Universitaire d'Informatique, Université de Genève, 207–245.

[Papathomas 95] PAPATHOMAS, M., 1995. Concurrency in object-oriented programming languages. In Nierstrasz, O., Tsichritzis, D., Eds.: *Object-Oriented Software Composition,* Prentice-Hall, 31–68.

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

[Philippsen 95a] PHILIPPSEN, M., 1995. Imperative concurrent object-oriented languages. TR-95-050, International Computer Science Institute, Berkeley.

[Philippsen 95b] PHILIPPSEN, M., 1995. Imperative concurrent object-oriented languages: an annotated bibliography. TR-95-049, International Computer Science Institute, Berkeley.

[Robert/Verjus 77] ROBERT, P., VERJUS, J.-P., 1977. Toward autonomous descriptions of synchronization modules. *Proc. IFIP Congress 1977,* North-Holland, 981–986.

[Rosenberry et al. 93] ROSENBERRY, W., KENNEY, D., FISHER, J., 1993. *Understanding DCE,* O'Reilly.

[Rozier 92] ROZIER, M., 1992. Chorus. *Usenix Int. Conf. on Micro-Kernels and Other Kernel Architectures*, 27–28.

[Schill/Mock 93] SCHILL, A., MOCK, M. 1993. DC++: Distributed object-oriented system support on top of OSF DCE. *Distributed Systems Engineering 1(2)*, 112–125.

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

[Schröder-Preikschat 94] SCHRÖDER-PREIKSCHAT, W., 1994. *The Logical Design of Parallel Operating Systems*, Prentice-Hall.

[Sheffler 96] SHEFFLER, TH.J., 1996. The Amelia Vector Template Library. In [Wilson/Lu 96], 43–90.

[Skjellum et al. 96] SKJELLUM, A., LU, Z., BANGALORE, P.V., DOSS, N., 1996. MPI++. In [Wilson/Lu 96], 465–506.

[Stroustrup 93] STROUSTRUP, B., 1993. *The C++ Programming Language.* Addison-Wesley.

[Sun 95] SUN 1995. *C++4.1 Library Reference Manual, Section 2.* Part No. 802-3045-10, Nov. 1995, Sun Microsystems Inc.

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

[Thomas 94] THOMAS, L., 1994. Inheritance anomaly in true concurrent object-oriented languages: a proposal. *IEEE TENCON '94,* 541–545.

[Thomsen 87] THOMSEN, K.S., 1987. Inheritance on processes, exemplified on distributed termination detection. *Int. J. of Parallel Programming 16(1),* 17–52.

[Tokoro et al. 92] TOKORO, M., NIERSTRASZ, O.M., WEGNER, P., EDS., 1992. *Proc. ECOOP '91 Workshop on Object-Based Concurrent Computing,* LNCS 612, Springer-Verlag.

[Tomlinson/Singh 89] TOMLINSON, C., SINGH, V., 1989. Inheritance and synchronization with enabled-sets. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89), ACM Sigplan Not. 24,* 103–112.

[Tripathi/Aksit 88] TRIPATHI, A., AKSIT, M., 1988. Communication, scheduling and resource management in SINA. *J. of Object-Oriented Programming 1(4)*, 24–41.

[Wegner 90] WEGNER, P., 1990. Concepts and paradigms of object-oriented programming. *ACM OOPS Manager 1(1),* 7–87.

[Weihl 89] WEIHL, W., 1989. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. on Programming Languages and Systems 11(2)*, 249–283.

[Wilson/Lu 96] WILSON, G.V., LU, P., EDS., 1996. *Parallel Programming Using C++,* MIT Press.

[Wing 94] WING, J., 1994. Decomposing and recomposing transaction concepts. In [Guerraoui et al. 94], 111–122.

[Wolff 95] WOLFF, TH., 1995. Transparently distributing objects with inheritance. *Proc. 28. Hawaii Int. Conf. on System Sciences,* IEEE, 222–231.

[Yokote/Tokoro 87] YOKOTE, Y., TOKORO, M., 1987. Experience and evolution of Concurrent Smalltalk. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications OOPSLA '87), ACM Sigplan Not. 22(12),* 406–415.

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

[Yonezawa et al. 86] YONEZAWA, A., BRIOT, J.-P., SHIBAYAMA, E., 1986. Object-oriented concurrent programming in ABCL/1. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86), ACM Sigplan Not. 21(11),* 258–268.

[Yonezawa/Tokoro 87] YONEZAWA, A., TOKORO, M., EDS., 1987. *Object-Oriented Concurrent Programming*, Computer Systems Series, MIT Press.

[Yonezawa et al. 93] YONEZAWA, A., MATSUOKA, S., YASUGI, M., TAURA, K., 1993. Implementing concurrent object-oriented languages on multicomputers. *IEEE Parallel and Distributed Technology,* May 1993, 49–61.