Issues in Object-Oriented Concurrency

Cristóbal Pedregal Martin*

18 December 1988

Abstract

The work described in this paper is a first attempt to find a synthesis of concurrency and the object model.

A representative sample of concurrent object-oriented languages has been analyzed to identify issues –dimensions– peculiar to the conjunction of the two features of interest. The presentation includes sections that review and develop the basic concepts, both in concurrency and the object model, needed for the analysis of the languages.

The relevant issues are presented in a structured way, along with a discussion of pros and cons of the possible alternatives. Issues identified include several concurrency and encapsulation features and also communication, migration and transparencies among others.

Some preliminary conclusions are offered, as well as suggestions for future work. The paper ends with an extensive list of references.

*Report of Graduation Work (**"Trabajo de Grado"**) at ESLAI (Escuela Superior LatinoAmericana de Informática) under the supervision of **Carlo Ghezzi** and **Norma Lijtmaer**.

1 Introduction

This work is an attempt to find a synthesis of two important elements in the design and construction of software, as they are reflected in programming languages. We investigate object-based programming languages that offer concurrency (henceforth also called concurrent object-oriented languages.)

concurrency The design and construction of software in which (loosely- or tightly coupled) concurrency is involved has always been a hard task. In the past, it was reserved to the initiated who were responsible to build certain type of software, such as operating systems and some real-time or distributed applications.

The discipline evolved, and new tools and concepts were added to ease the task of the concurrent programmer and improve the quality of her work. Unfortunately, these achievements have not kept pace with the increased complexity and diversity of demands that are put on concurrency. Driven mainly by the desire to take advantage of decreasing costs of hardware, but also for more stringent reliability (and other) constraints, the constant need to tackle more complex problems, and especially problems whose solution is expressed naturally in concurrent terms (thus avoiding overspecification), the discipline of building concurrent programs is now spreading to areas and (uninitiated) people not expected a few years ago. The quest for new design techniques, new paradigms, etc., has increased correspondingly, but we are still far from a panacea.

object model Though the object-oriented model has roots in simulation and artificial intelligence, they remained relatively unknown until the eighties, when it sprang to notoriety with the Smalltalk phenomenon. The uniformity of the approach ("everything is an object or a message between objects") captured the interest of a community burdened by the complexity of the problems it was tackling and the tools it was using.

To name just two of the expectations raised, it was anticipated that programming would move closer to design, and that reusability of software would be practical. While initial results have not been miraculous, there are signs that indicate that this is a (maybe 'the') right approach to building software.

Object-oriented methodologies are having a deep effect in the culture and practice of software building, and we witness trends to introduce them in all kinds of application domains, at least as a new engineering paradigm, much in the fashion of 'structured programming' in the seventies.

concurrency + objects It is not surprising, given the importance of both concerns, that several groups of researchers trying to apply the object-oriented methodology to the harnessing of parallelism. This is done through attempts to extend the object-oriented paradigm to include concurrency. Besides, for reasons similar to those that fuel the use

of concurrency, there is a great demand for distributed languages and systems, so the initiatives tend to include both concurrency and distribution.

This works tries to contribute to a better understanding of the nature and underlying assumptions of object-oriented concurrency. We began our work by surveying the literature, selecting a set of six languages to study: ABCL/1, Argus, Emerald, Hybrid, NIL, and OTM. Foundations are considered first, and only then is presented the core of the work, consisting of a set of interesting features identified during our survey of the languages.

We analyzed the literature for each language to find out how (and if) the facilities associated with concurrency and object-oriented are provided. This seemingly simple task turned out to be difficult, for the terminology is very heterogeneous and sometimes misleading. Just as an example of this, the notion of 'thread' (of control) has about as many meanings as authors, and we had to resort to operating systems literature to give it a consistent meaning. It was mainly the recognition of this fact that influenced the subsequent work, for we felt the need of a framework to clear this up (hence the first sections on foundations became essential to agree on both *terminology* and *concepts*.)

After the features that are characteristic of this class of languages were charted, we set out to understand their relationships; essentially how the features contribute to providing concurrency and object-orientedness, and how concurrency and object-orientedness were constrained by the realization of the features.

The preceding led us to identify the features that are peculiar to this class of languages. By 'peculiar' we mean the features that provide or reflect the two qualities of interest. The rest of the features, which remain unaffected, are orthogonal and thus irrelevant for this analysis.

structure of the paper This work is organized in the following way. Section 2 contains a discussion of the main issues concerning concurrency; it examines basic concepts and language constructs, and ends with a discussion of the concept of thread.

Section 3 deals with objects. Beginning with an informal discussion (in which the notion of data abstraction is introduced), it uses a classification scheme to identify interesting features of object-oriented languages. Concurrent features are examined with the aid of concepts introduced in section 2.

Section 4, which discusses relevant features in the class of languages of interest, constitutes the core of the paper. It starts with some remarks on the scope of the analysis and an explanation of the structure adopted to present the features. It then enumerates and defines the main issues. These deal with concurrency, communication and synchronization, encapsulation, atomicity, fault tolerance, and transparencies. Concurrency and encapsulation contain several sub-issues; transparencies goes a little farther than the rest.

Section 5 gives some concluding remarks, and suggests future work. It is followed by the acknowledgements.

The paper ends with an extensive list of references.

2 Concurrency

Concurrency corresponds to the everyday experience of carrying out several tasks at the same time, something typical of human activity; as [Hogg 87] puts it:

> we note that humans can simultaneously chew gum and walk in a straight line

But things are not so simple in the world of computer programming. When studying programming, the conventional approach is to avoid concurrency as long as possible, trying instead to find sequential algorithms to solve the problems. This is attributed to the increased complexity inherent in dealing with concurrent algorithms. It is hard to 'think concurrent,' to use a loose but evoking expression. Not to mention the difficulties associated with specifying, deriving, and proving correct concurrent programs (as compared with the same activities in the case of sequential programs).

However, there exist several strong reasons to use concurrent algorithms. One is economic: the more parallelism one has, the more (computational) resources can be poured in to solve (faster) a problem. Trends in the recent past show that, with the declining costs of hardware, the primary efficiency concern for a large family of applications is execution time (correctness provided, of course: more on this below). Since physical limits to computer speed are fast approaching [Kung 88], it is clear that introduction of parallelism is the only way left to speed things up.

The other reason is –conceptually– more interesting: many problems are inherently concurrent in their formulation. The interest for concurrent solutions is obvious: they will be the most natural ones! To understand why this does not contradict what was said above about the complexity of concurrent solutions the reader should realize the gap between the 'nature' of a problem and its formalization in a way suitable for the derivation of a computing solution.[1] And here is where programming languages play their part: they are –at some level– the tools the problem solver uses to express her understanding of the problem, or better, the tools she[2] uses to express a suitable algorithm.

It is not within the scope of this paper to discuss the precise role of programming languages in problem solving or even in the formulation or description of algorithms. We nevertheless believe they are important enough to justify their study. Moreover, in dealing with object-oriented concurrent languages, the field is very far from closed [Moss 88].

At this point, we hope to have motivated the reader to carry on, ready to endure the review of concepts related to concurrency. We believe this section to be necessary because of the myriad definitions, semantics, etc., through which concurrency is introduced in the literature. The following is not a complete survey, but should suffice as a recapitulation and to agree on some basic concepts and terms.

[1] Notice that a "solution" may well be a simulation or other non-functional activity.

[2] Or 'he.' Following [Knuth 69], this should not be taken too seriously.

2.1 Fundamental Concepts

We do not start from the definition of algorithm, program, process (these issues well dealt with in e.g. [Ancilotti 88]) but instead review some basic concepts.

The (computational) *activity* is the basic notion of (sequential) execution: it may be conceived as abstracting the notion of one processor executing one program (or as the dynamic counterpart, shown in the trace, of the static specification represented by the program.)

We say that two activities `A` and `B` are *concurrent* if they *do not have* a necessary temporal *ordering* between themselves, i.e. `A` might occur completely before `B`, the reverse could also happen, or they could even overlap in time. (An intuitive notion of time suffices for our purposes.)

We have *parallelism* when concurrent activities overlap in time; we also say in that case that concurrency is being exploited.

Two concurrent activities *communicate* when they are able to exchange information by some of the means described below.

2.2 Language constructs and tools for concurrency

Various constructs to express concurrency in programming languages have been proposed. We will briefly review them in the following, only to refresh some terminology and concepts: this is neither a primer on the subject, nor an extensive or critical review. Its purpose is just to set the stage for the discussion of the issues considered in section 4.

Since the material is quite established, we will omit citations, referring instead the reader to sources of references. Such sources are a textbook on concurrent programming [Ancilotti 88] and a textbook on operating systems, [Peterson 85]. Both have very good bibliographic notes; the former has a very thorough treatment of all the language constructs while the latter provides a more informal but very clear description of the main constructs.

2.2.1 coroutines

The first construct to handle concurrency at a language level is the coroutine, introduced by Conway (1963). A coroutine is like a procedure in that it has local declarations and code. It differs from a procedure in the way control is handled, and in the fact that its state persists across control transfers.

A procedure is invoked with a *call* statement and returns control to the caller via a *return* statement. There exists a hierarchical relation between caller and callee: the callee does not know which unit called it, and cannot choose where the control will flow to on its termination.

A coroutine, on the other hand, may suspend its execution via a *resume* statement, transferring control to another coroutine whose name is indicated as the parameter in the resume call. Coroutines are therefore organized in the same level whereas procedures are

organized in a hierarchy. When it is resumed later, the coroutine continues execution from the point it did its last resume, its state unaltered.

Notice that the coroutine concept allows the interleaving but not the parallel execution of activities, i.e., no two coroutines can be executing simultaneously. Considering uniprocessor machines, however, they are very useful, for any concurrent program is executed by interleaving, so coroutines are a good basis on which to build an interleaved implementation of concurrency.

[Doeppner 87b] points out that while coroutines are independent units of execution, the different concepts of synchronization and scheduling are merged in one construct: the resume. That is, one cannot separate these notions because whenever control is relinquished it is necessary to specify to whom it is passed. A better mechanism would allow the choice of the next unit to be executed independently of the control transfer. This remark should be seen in the light that coroutines are a fairly low level construct –in fact they reflect what happens in a multiprogrammed uniprocessor. Besides that, it is fairly simple to separate both concerns with the addition of one coroutine that works as a scheduler. Then all the coroutines are programmed to resume the scheduler, which in turn chooses and activates the next coroutine.

2.2.2 fork and join

The *fork* and *join* pair of constructs is also due to Conway. The *fork* instruction produces two concurrent executions within a program, namely the continuation of the invoker and the spawning of a new one starting from some point in the program (specified as a label in the argument), thus splitting in two the control flow. Note the difference with coroutines: we now have two executions that are rather independent, but are in fact executing (portions of) the same code, and sharing the name space.

The *join* is the symmetric operation. When encountered by an activity, it finishes its independent execution and disappears. Note however that its parent activity (the one that *fork*ed it) anyway 'inherits' whatever data the dead activity might have elaborated for these data is in the shared name space. So an explicit passage of results is not needed.

This is a powerful primitive: it can be proved any form of precedence in concurrent actions may be expressed as a composition of forks and joins.

2.2.3 parbegin ... parend

This parallel construct was introduced by Dijkstra. It is analogous to an Algol block in its shape. But in this case the statements enclosed in the brackets are all executed concurrently. It is more structured than the fork and join, but less powerful. There are precedence constraints for concurrent programs which are not representable with *parbegin... parend* that are representable with *fork... join* (see example in [Peterson 85]). But the power is equated if the par- pair is augmented with suitable synchronization primitives (e.g. semaphores).

2.2.4 processes

Processes are another way of structuring concurrency. The idea is to allow several sequential programs to execute concurrently, each having its own program counter, name space, etc. Interaction among them may take essentially two shapes: by message passing or by sharing of variables [Lauer 78].

Much is gained in this schema, for encapsulation allows certain conflicts (e.g. interference on shared data) to be managed more easily. Also, the design and programming are easier –at least in principle– because the smaller units are sequential: concurrency only occurrs among full-sized processes (i.e. processes the size of stand alone programs).

Their drawback is the overhead: creating and swapping processes means moving a lot of information. (In some earlier systems with this model, processes were statically created [Brinch Hansen 78].)

2.2.5 synchronization

There exist several ways of synchronizing activities: they are roughly divided according to whether there is any memory shared among the activities.

In the case of shared memory we have the *semaphore* due to Dijkstra, which is a variable that may be accessed only through two special operations, **P** and **V**. The semantics of the operation is such that under certain conditions an activity executing a **P** on a semaphore may be delayed. It will only allowed to carry on when another activity executes a **V** on the same semaphore. So its functioning reminds us of the system for crossing a narrow (one-lane) bridge, in which the driver of the last car of a group allowed to proceed is given some token (e.g. a colored stick) on one end of the bridge, and only when she reaches the other side is it possible to allow the crossing in the other direction.

It is well known that semaphores are elemental and powerful, but very unstructured –thus prone to error.

Another, more structured synchronizing primitive is the *monitor* construct introduced by Hoare. It encapsulates data so that it is manipulated only through exported operations, and its semantics guarantees that there is at most one active operation at a time, thus serializing access and preventing interferences in the shared data.

In the case of no shared memory, synchronization is done via synchronous message passing. Sending (or receiving) a message is viewed as calling a special procedure, in which the activity is stuck until the partner of the synchronization does the symmetric operation. Then both return normally, resuming their respective executions.

2.3 Threads: a controversial concept

Consider a conventional operating system process. It is a dynamic concept, in that it is the executing counterpart of a static specification of behavior given in its corresponding program. It is basically an executing entity, with associated resources and data. It comprises

several parts. It has a name space that delimits the entities it owns, as well as permissions to access and use resources in the system. It is protected by boundaries from other processes. It has an execution state (i.e., ready, suspended, etc.) It may have an associated priority. But, again, it also has a dynamic part, i.e., a thread of control, which leaves as a trail the 'trace' representing the path of execution. This thread is like an abstraction of the processor, whereas all the environment corresponds to the physical resources, e.g. memory, peripherals, etc., the processor controls (analogous to the 'owning' of objects in the name space of the process.)

As suggested in [Rashid 88], we may split the process abstraction into two parts. One is the *task*[3] part of the process, which is the basic unit of *resource allocation*, and abstracts the resources the executing agent has access to.

The other is its *thread*, the basic unit of *cpu utilization*, or abstraction of the executing agent, which corresponds to our intuitive concept of activity, outlined above.

But consider now having more than one thread in the same environment: now we have concurrency 'internal' to the process, sharing the same name space. In our machine analogy, it is as though we had a multiprocessor with a shared central memory.

We now have a notion of a process which may have internal concurrency through the existence of several threads of control sharing an address space (*multithreading*). This of course requires adopting precautions to prevent the interference typical of unchecked concurrency, i.e., the problems derived of sharing entities under parallelism.

These ideas have been introduced and studied exhaustively in the operating systems literature. The preceding definition is inspired mainly in the ideas of Mach, but similar ideas are found in Topaz [McJones 87], which strives to offer multithreading in Unix processes. A process is defined there as a set of threads of control executing within a single virtual address space. Also interesting is the system *Threads* [Doeppner 87a, b]) which uses threads as the mechanism for obtaining cheap concurrency.

The main point of agreement among these similar notions of thread is their low cost and ability to express concurrency (see discussion on light-weight and fine-grain processes in section 4). When it comes to explain things in more detail, however, the lack of a suitable, unifying computation model is felt. Explanations of 'why' threads abound; it is when confronted with the 'what' that things get fuzzy.

Threads are explained in very low level implementation terms (e.g., a thread is a program counter with an associated stack [Wegner 87], or maybe even without the stack and local variables [Doeppner 87b], etc.)

Another point concerns to where threads belong. To some, threads can be passed around as message requests, and of course queued up waiting for a service [Wegner 87]. Nierstrasz views a thread as a token, which traverses all the units (same as Wegner's) of the program indicating which is active at any given time [Nierstrasz 88]. Thus a thread may cross object and procedure boundaries (i.e. across name spaces) to carry out its job, always incarnating

[3]So called in the mentioned work; we use 'task' for lack of a better alternative. Maybe 'environment' or 'envelope' conveys the concept better.

the execution of the program.

To others –e.g. the designers of Argus [Liskov 82, 88]– the picture is different. In Argus, a thread belongs to the guardian in which it runs. Moreover, a thread may go out to request a service in another guardian, but *by definition* it cannot enter it: it is delayed waiting for the completion of the service. Inside the guardian a new thread is created with the sole purpose of honoring the request. This discourse is also used in Ada when a server is implemented with a *task* module. Although in this case the task is sequential, we say that its internal thread is suspended in a select statement, waiting for an adequate message to arrive. (The fact of the existence of a queue in which requests are delayed –Ada– or the spawning of new threads –Argus– is not important here: what matters is that in these cases threads do not enter foreign objects.)

It should be mentioned that threads are also named *lightweight processes* by some authors; in fact this may be their earliest name [Lampson 80, Jazayeri 88].

3 Objects

This section reviews the concept of object-orientedness in the domain of programming languages. It first introduces the notion of object in an intuitive way, through an informal discussion. Then a classification scheme is offered, aiming to identify interesting features of object-oriented languages. Concurrent features are examined using concepts from the preceding section.

3.1 What is an object?

In the real world, an object is "anything with a crisply defined boundary" [Cox 86]. While this is certainly not enough to work with the object concept, it reflects the important things about objects in the computer science 'world.' One of the essential reasons why people want to use object oriented methodologies is because they want to map things they see in the real world into computer representations. This approach is feasible not only in simulation, where its usefulness is obvious,[4] but also in many situations in which a problem, its solution, or both can be thought of in terms of objects.

An Object is essentially an encapsulation which encloses some data –its *internal state*–, offering manipulation of this data only through the use –*invocation*– of a prescribed set of operations –its *interface*.

The concept of encapsulation as we use it nowadays recognizes several ancestors, from the (weak) module offered in Fortran, Simula's classes, to CLU clusters [Liskov 86], Smalltalk objects [Goldberg 83], Ada packages and tasks, Modula-2 modules [Wirth 82], etc. Besides programming languages, other examples may be drawn from fields such as artificial intelligence, notably the Actor model [Hewitt 77, Agha 84].

The primary goal of encapsulation in (current) programming languages is that of realizing the notion of *data abstraction*. It has been the subject of much research; a good introduction is [Liskov 74], but it recognizes antecedents at least back to Simula 67 (see Dahl and Hoare in [Dahl 72].) While alternative meanings for 'object' may be found, we use the primary notion of abstract data type in our analysis.

Liskov and Guttag in [Liskov 86] give the following definition of abstract data type:[5]

> The behavior of data objects is expressed most naturally in terms of operations that are meaningful for those objects. This set includes operations to create objects, to obtain information from them, and possibly to modify them. For example, *push* and *pop* are among the meaningful operations for stacks, while integers need the usual arithmetic operations. Thus a *data abstraction* (or *data type*) consists of a set of objects and a set of operations characterizing the behavior of those objects.

[4] And it all began, with Simula [Dahl 72].

[5] We will use abstract data type or data abstraction interchangeably.

(It is understood that data may be accessed only through the operations provided.)

The behavior of an abstract data type may be formally specified (e.g. algebraically as proposed in [Guttag 78].) Formal specification means not only giving the types of arguments and results of the operations (i.e. the *signature*) but also their semantics, i.e. their behavior.

An interesting alternative description of abstract data types viewed as 'machines' with a state and operations is suggested in [Meyer 88]. The phrase 'active data structure' used there suggests the idea of machine, but it has nothing to do with the meaning we give to 'active objects' in this work.

3.2 A classification

There exist several quite varied views on what is the essence of object-oriented programming (and design). The extent of the differences we is apparent in that there still is disagreement concerning whether inheritance is an essential feature or not. Some authors, like Goldberg, Wegner, and Meyer consider it must be provided for a language to be considered object-oriented. Others, however, like Cox and Gehani, consider it is not mandatory.

Since we are more interested in reviewing several different languages –whose existence precedes and ignores any classification– we will be less strict than we would be if our goal were to define 'the' concurrent, object-oriented language. Available classifications (e.g. [Wegner 87, Meyer 88]) identify a set of features and term 'object-oriented' the languages that possess a certain specified subset of features. Other authors stick to their languages (e.g. Cox with Objective-C and Goldberg with Smalltalk) and do not bother to offer a framework in which to situate them.

Meyer does the classification as seven successive 'levels,' only calling object-oriented those languages that reach the last level. Needless to say, Eiffel, his language, qualifies neatly. Some interesting points in his proposal are the requirement of automatic memory management, and the requirement of multiple inheritance. (These terms will be defined shortly.) But the issues of concurrency and distribution are not contemplated; they are not regarded as independent –orthogonal– either. That is, these topics are not considered in Meyer's analysis. Maybe this shortcoming comes from the fact that the discussion is centered around Eiffel (which is an excellent language otherwise). It is for these reasons that we do not use this as a framework.

We prefer Wegner's classification [Wegner 87] instead, mainly because it includes concurrency as a relevant dimension. We only use it to establish some terminology and allow our work to be related to other work.

We review Wegner's classification in two parts, namely the nonconcurrent issues first and then the possibilities of concurrency in a somewhat greater detail. The following two subsections are adapted from [Wegner 87].

3.2.1 Non-concurrent features

The analysis identifies six orthogonal dimensions of object-oriented language design. Concurrency is dealt with in the next subsubsection; the rest are explained here.

class classes serve to classify objects in sets with uniform behavior. They specify operations common to all instances and serve as a template from which objects may be created.

inheritance class inheritance is a mechanism for sharing operations defined in a superclass by a number of subclasses. Inheritance schemes differ in the way an invoked operation of an object is matched to a definition.

data abstraction it is an object whose state is accessible only through its operations. Its state is generally represented by instance variables.

strong typing a language is strongly typed if type compatibility of all expressions representing values can be determined from the static program representation at compile time.

The work further proposes some names associated to languages that offer certain subsets of the features just defined. We will not examine those because we do not require that much; we accept a much looser meaning of object-oriented. In particular, though we agree in that inheritance is essential, we drop that requirement in the case of the languages we study, essentially because not doing so would force us to switch names all the time, complicating the task of going from our work to the sources and vice versa (because of the additional 'level of indirection' imposed in the names).

We are interested in the concurrency part of Wegner's classification, and it is important to situate it in the context of the whole classification.

3.2.2 Concurrent features

Before examining the possible forms of concurrency in object-based languages, let us introduce the central entity with which we will deal in the rest of the work, namely the *processes* (which we will call *active objects*.)

active objects active objects have an object-like interface of operations and one or more threads of control that may be active or suspended. (Threads were defined in the section on concurrency.)

There are two aspects in this classification. One is how is the internal concurrency is provided; the other concerns ways of synchronizing the threads.

internal concurrency three classes of internal concurrency are identified:

sequential (single thread) Sequential processes contain only one thread and queued entry points, where the requests wait until the process gets free to serve them. Note that if the thread is suspended –e.g. waiting on an I/O condition– the process is blocked and no one else is served until the current request is completed. Ada and NIL [Strom 84] offer this semantics in their active objects.

quasi concurrent These active objects allow active threads to be suspended placed in condition queues to be activated when some other thread causes the condition to change. When the active thread terminates or is suspended, a waiting thread (either from a condition queue or the entry queue) is selected to become active (typical monitor semantics). The result is that there is at most one active thread within the object, but the time zero threads execute are reduced, as compared with the sequential case. ABCL/1 presents a behavior somewhat like this.

concurrent (multiple threads) No restriction is placed on the number of threads that may be active simultaneously inside the object. Naturally some tools for managing shared access, etc. are provided (e.g. locks, atomic objects). In this way threads are not obstructed to enter the object; synchronization is imposed at the internal data objects level. Argus [Liskov 82, 88] is an example of this class.

synchronization from an abstract viewpoint two general classes of synchronizations appear:

thread with thread this is the direct synchronization (obtained for example through a rendez-vous). It involves the identity of both threads.

thread with data this of course is a form of synchronizing threads (passive entities are not synchronized), but indirectly through data. This happen when grabbing a mutex lock that grants access to some data, which forces other threads to be suspended waiting for the lock.

Remark: This classification should be taken only as a guideline to frame the issues.

4 Concurrency, distribution, and objects

In this section we examine relevant issues in concurrent distributed object oriented languages. We will use the background laid out in the preceding sections to understand the features presented here. We believe the following features constitute what characterizes a language of the class we are considering. We have based our analysis in three main sources:

- a set of recent languages that have been branded 'concurrent object-oriented' in the literature.
- other well known languages –mainly Ada and Mesa [Lampson 80]
- some (distributed) systems, such as Accent/Mach [Rashid 88], Topaz [MacJones 87], Threads [Doeppner 87a,b]

Before going into the enumeration and analysis of important aspects of languages, it is important to stress the limitations of the analysis: we use an informal method. A central idea is the concept of consistence, but no claim will be made that the proposed set of language features is consistent. Moreover, this is not true of the set of all proposed features; some of them are mutually conflictive. (In that case a tradeoff is obtained using more specific criteria derived for example from the application domain.) We are not be too concerned with this because we are not trying to design a language, so we do not need to choose which features to include in it, and at what price. Note that claiming consistency of a set of features would entail exhibiting a language which contains all of them. This, if one follows [Wegner 87] in his definition of consistence and orthogonality. But it is perhaps better to define it in terms of a model that describes the basic elements needed to construct a language of the class we are interested in. (Wegner's definition relies on the exhibition of case languages, but a case language may be poorly designed.) So we leave this question open.

We nevertheless believe the analysis to be very useful, both to our present purposes of gaining a better understanding of concurrent object oriented language and in the quest for a formal treatment of them. We think that a first approximation to this problem should be informal and somewhat experimental.[6]

While we do not claim to have defined a design space in a rigorous sense, we have isolated sufficient but maybe more than necessary concepts to start trying an identification of 'elemental' ideas. We expect criticism to our 'shopping list' of features. On the other hand, the notion of design space has the drawback that all dimensions are born equal, i.e., we are forced to adopt a flat hierarchy of essential features. While this is desirable from a formal (and maybe aesthetic) point of view, we think that it does not reflect our current knowledge of the area. Only when we identify the very basic concepts on top of

[6]Taking the languages and their features as phenomena to observe and try to explain should ultimately lead to building a useful model.

which everything can be built[7] will it make sense to consider such 'principles" of equal importance.

For the time being, we will make do with weaker devices such as the structure of subsections and paragraphs to convey what we perceive as a reasonable hierarchy and relative importance of concepts.

4.1 Three concurrency issues

One natural aspect to look at in these languages is that of concurrency. We are interested in the way in which concurrency is introduced (or offered) in the language.

There are three aspects to concurrency here. The first is whether the semantics of the language prescribe an interleaved execution (nevertheless called 'concurrent') or allow true concurrency.

The second is peculiar to concurrent object-oriented languages and is about the relation of objects to concurrency: is concurrency offered between objects (i.e. by concurrent execution of different objects), within objects (i.e. by allowing several threads inside an object), or both?

Finally, we have an issue that also arises outside the object-oriented domain[8] but is strongly coupled to the previous, namely the 'grain' and 'weight' of concurrency offered by the language.

4.1.1 Concurrency: 'true' versus interleaving

It has already been pointed out that some constructs (e.g. coroutines) allow (quasi) concurrency but not parallelism. In particular some models of (active) objects can service only one request at a time, i.e. can have at most one active thread. This is the case in for example NIL [Strom 84] and ABCL/1 [Yonezawa 86]. These languages are nonetheless termed concurrent because they allow interobject concurrency. Quasi concurrency is useful because it reduces the time that zero threads are executing [Wegner 87] (when a thread suspends itself to wait for an event, it allows another thread to enter the object).

Other languages, in contrast, offer real parallelism. Such is the case of OTM (with its *reflex* construct, roughly a fork) and Argus [Liskov 82, 86, 88].

In Argus two mechanisms, one implicit and the other explicit, provide concurrency. The implicit mechanism is given by the fact that a new thread is created ('spawned') for each request thus allowing concurrent execution of different calls, possibly different instances of the same service. Note that this device conceptually eliminates queues (although queueing does happen whenever the number of requests exceed that of available processors –but this is and implementation issue and is taken care of by the runtime system).

[7] I.e., in an 'axiomatic' sense

[8] For instance, in (distributed) operating systems, e.g. Mach [Rashid 88], or concurrent languages, e.g. Mesa [Lampson 80]

The explicit concurrency construct is the *coenter*, which behaves like Dijkstra's parbegin/parend pair.[9]

4.1.2 Intra- and inter-object concurrency

Concurrency may be obtained in an object environment in two different ways, namely:

intra-object (or intranode) it is when inside an object several threads are allowed to execute concurrently (this may turn up to be quasi-concurrent, like e.g. in ABCL/1)

inter-object it is simply due to the independence of different objects, which may be executing simultaneously. This case is simpler because by the very nature of these objects (i.e. because they are units of distribution) there is a very controlled interaction among them because they do not share any space and communicate through a well-defined interface.

Some authors [Power 88] argue that there is a third form of concurrency, namely that obtained by creating an object. We consider this to be a special case of internode concurrency, because concurrency is increased by independent and loosely coupled execution of new threads in a different name space. Power's view does not affect essentially our subsequent discussion: the reader may take either view and transform the statements suitably and our discussion will retain its meaning. It is important to point out that all languages of the class we consider allow dynamic creation of objects (something essential to object-orientedness [Wegner 87]), so we will discuss this no further. Similarly, no explicit deallocation is allowed, the runtime system being responsible for reclaiming unreferenced objects.

Regarding intraobject concurrency we note there are several possibilities. One is to have a thread continue after the return of a service. (This is called 'postaction' in Hybrid, see [Nierstrasz 87].) An example of this might be the balancing of a tree after an insertion, with the service finishing after the insertion, and the (active) object doing the balancing on its own afterwards. After the return there is concurrency between caller and callee.

Another example is that of having 'background' processes, i.e., processes that execute continuously without interfering with the services the objects provide. Their task may be 'housekeeping' (see [Liskov 82]) of some sort, and they are usually started on object creation.

Finally, the most general case is that active objects possess their own thread, being free to carry out their activities even when not replying to any message. Contrast this to the sequential (e.g. Eiffel [Meyer 88]) case in which there is only one thread active at any given time, and the activity passes from caller to callee on a request and back to the caller on a return. In this case, there is some initial object that starts the activity of the whole system by invoking others, and this cascades. (The fact that on creation some initializing code of

[9] Argus offers in fact more power: it combines these with atomic actions.

different objects might be concurrently executed does not bear relation to this observation, for it only concerns the establishing –initialization– of the invariant for the internal state of the object.)

There is a subtle point on which there is no agreement: does a thread cross object boundaries? According to [Nierstrasz 88], a thread (like a token) goes from object to object, indicating which is in an active state at each moment. So when object `A` makes a remote procedure call (RPC) to object `B`, the thread leaves `A` and enters `B`.

In Argus (as described in [Liskov 82]), the calling thread is suspended awaiting the outcome of the remote procedure call, while a new thread is created at the called object. Note that the creation of the thread is not essential here; we are concerned with the fact that threads are tied to fixed environments (i.e. their object's), and so do not cross object boundaries by definition. The argument can be recreated considering an Ada task with a select construct to accept requests. When no request is pending, the thread internal to the task is suspended on the select. When a request arrives, this inner thread is activated and the requester thread may be thought of either merging with the server thread or waiting outside for the results.

4.1.3 The 'grain' and 'weight' of concurrency

The ideas presented in this part are not independent of those in the rest of this subsection, but we felt it would be more clear to deal with them separately.

The notion of *grain* reflects the fact that the size and structure of the program (or program unit) which gives origin to the concurrently executing threads may range from rather small and simple to large and complex. Small grain concurrency, then, means that fine details of concurrency can be expressed separately, whereas large grain groups some activities together in a bigger sequential program. In the latter case some of the concurrency intrinsic to the problem is lost; in other words, lack of sufficiently fine grain of concurrency imposes constraints (i.e. by forcing sequentialization) that are alien to the nature of the problem.

Since a concurrent specification defines a partial order that the computation must respect, what happens is that total orders are imposed on individual actions until 'lumps' of size the smallest available grain of concurrency are obtained. These 'lumps' are then allowed to execute concurrently (but the partial order is now 'poorer,' i.e. it has fewer possible paths than the original had.)

To gain a better understanding of the idea of grain, consider a loop operating on a matrix in a conventional program. The purely sequential approach would represent the coarsest grain (i.e., a total order is imposed on the operations to be carried out). Then one might decompose the matrix into rows and treat them concurrently, with each row being dealt with sequentially. Finally, one could go as far as evaluating subexpressions concurrently.

That is a rather artificial example (it is likely that all cases would be branded 'fine grained' in the literature), but it shows the point. It is clear that the finer the grain, the

more concurrency can be exploited. In real languages, it is accepted that a computation that involves the execution of a few high level statements (one of the coarsest grains in our example) qualifies as fine grain, whereas a conventional operating system process is coarse grain. It is the associated information as well as some sort of length what comprises the grain.

The idea of the *weight* of a process is closely related to that of grain. Indeed, some authors use lightweight and fine grain interchangeably. It is not our intention to settle this now; we will content ourselves with pointing out an additional issue that is worth taking into account. The main interest in lightweight processes (also called threads) is 'economic': the only agreement is that they are 'cheap', i.e. easy to create, destroy, and swap. This is subtly different from the grain, having less to do with how much concurrency it may model than with how complex (heavy) the context is (i.e. local variables, access rights, protection, and everything that need to be associated with it; therefore need be created, destroyed, swapped.)

4.2 Communication, synchronization

The types of communication mechanisms available to a language of the kind we are considering are essentially those usually associated with distributed systems. Though message passing is used in conventional object-oriented systems, it usually has the conventional procedure call semantics. (It is worth noting that in object-oriented languages message passing may be present without there being any distribution: consider Smalltalk [Goldberg 83].)

In the languages of interest, a diversity of mechanisms is offered, and while remote procedure call [Nelson 81] is widely used, there often is some asynchronous mechanism as well (e.g. in NIL, ABCL/1). Despite different names and appearances (see ABCL/1's 'future' objects), it amounts to a more or less structured asynchronous message passing. It should be pointed out that in the language descriptions little is said about the *semantics* of the mechanisms; in particular, consider the alternatives for RPC: 'exactly-once' (i.e. fail-free), 'last-of-many', 'at-most-once', etc.

Asynchronicity is a way to obtain concurrency, by allowing the requester of the service to proceed while the service is performed.[10]

When communication is synchronous, it also serves as a synchronization mechanism between different modules.

An interesting way to view synchronization mechanisms is by their relation to threads. Two classes are identified: those that synchronize threads with threads (e.g. rendez-vous) and those which synchronize threads with (protected) data (e.g. locks). Of course what are ultimately synchronized are threads, but indirectly, through the data they are trying to access.

[10] And even while the message is being sent as in ABCL/1, but we consider this to be dangerous: what about communication failures?

Another interesting (but in our view secondary) device present in several of the languages (e.g. ABCL/1) is the notion of *continuation*, also called forwarding or delegation (in OTM [Hogg 87]). Care must be taken with Hybrid terminology [Nierstrasz 87] which has a mechanism called delegation which is in fact a monitor condition primitive, i.e., nothing to do with forwarding.

4.3 Encapsulation

4.3.1 Data abstraction

The notion of data abstraction was introduced in the preceding section on objects (see Liskov and Guttag's definition in 3.1.).

Briefly, an *abstract data type* is a data structure with an associated collection of operation signatures (i.e. a specification of the types of their arguments and results), and the additional restriction that data may be accessed only through the exported operations.

4.3.2 Uniformity of the language: the objects.

A language may or may not be uniform with respect to the constructs it offers to define objects.[11] We notice two different trends here, which are best represented by Emerald [Black 86, Jul 87] (uniform) and Argus [Liskov 82, 88] and NIL [Strom 84] (non uniform). This is relevant not only from the point of view of distribution, but also concurrency, for the costs and types of intranode and internode concurrency are different. Notice that this –in principle– does not contradict that in an object oriented language all entities are objects as homogeinity is not a requirement.

uniform objects model: objects do not reflect distribution In the uniform model, objects, large and small, are programmed and offer the same semantics uniformly regardless of whether they are remote or local, shared or private (i.e. inside a bigger module). This approach is interesting from a conceptual point of view, and it has been explored in the Emerald language [Black 86]. In that particular case, some assumptions are made about the underlying system –the most important being that is homogeneous. But this is not unreasonable in a variety of situations –if only for economical, practical, or other non-semantic reasons. Their approach (explained in [Jul 87]) is quite reasonable and is probed further in the next section. But being the only initiative in that direction, it is difficult to assess its feasibility. A point to stress, anyway, is that even the model that Emerald offers is based on strong compiler support, which distinguishes virtual node objects when 'deciding' the implementation. In other words, the uniformity is obtained through a layer that hides the –quite natural, discussed next– virtual nodes model.

[11] Or other entities, but objects are the relevant entities in our case.

non-uniform objects model: virtual nodes are special objects In languages designed within this model, there exist two main classes of objects, of different "sizes." One is the usual data abstraction mechanism (i.e., a realization of an abstract data type) whereas the other, besides that, represents an abstraction of a node in a distributed system. In other words we have a module that encapsulates resources associated with a node, constituting a virtual node.

The intention is to convey some of the underlying architecture to the software designer, reflecting the different costs for tightly and loosely coupled cooperation.[12] The rationale is that since local communication is far cheaper, if the system decomposition (with respect to concurrency and distribution) is done right, the result is very efficient.

The approach seems reasonable in the current state of the art (i.e., granted the relative costs are as assumed). From an object-oriented perspective it has drawbacks. It can be objected that objects[13] become overloaded with the additional meaning of 'physical' encapsulation (virtual nodes).

The software designer may also criticize this approach. Consider a change of design decision[14] such as the following. After decomposing the system in modules A, B, and C (the language guarantees that they may reside on different nodes), the designer finds out that C may in fact be divided in two more or less independent parts (and so allocated on different machines to increase parallelism). Although the innards of C are composed of (small) objects, this is of no use to decompose C with respect to distribution and concurrency. The module must be redesigned.

The reverse may also happen, namely the wish to merge two virtual nodes. Note that this has nothing to do with any allocation policy the (language and runtime) system might offer. In the first example, the system will not allow the splitting of the module on two (real) nodes. In the second, some costs may be lowered by timesharing a node among the merged modules. But this solves the problem only partially: the language will still impose the restrictions associated with different name spaces and internode communication (e.g. it will not permit sharing of objects).

In sum, in the case the of heterogeneous model we have the tradeoff of getting more efficiency in return for early design decisions and a vision of the system closer to the underlying architecture.

4.3.3 Inheritance and distribution

Some authors view inheritance as an indispensable component of an object-oriented language [Meyer 88, Wegner 87]; others (e.g. [Cox 86]), consider it just desirable. Viewing

[12] This is due to the expense of crossing module boundaries and internode communication as opposed to intranode communication.

[13] Worse yet, only some objects.

[14] This is well embedded in the object-oriented philosophy, which caters to 'exploratory' programming, and frequently identifies programming with design [Goldberg 84, Meyer 88].

inheritance as a sharing mechanism[15] it conflicts with some modularity by hampering the encapsulation of parts of the system.

This conflict is manageable in tightly coupled concurrent systems, but is more acute in the distributed case: hen sharing becomes prohibitively expensive. According to [Wegner 87] distribution and inheritance are inconsistent; Liskov (in [Power 88]) believes it will be some time before costs are reduced to make them simultaneously feasible.

Aside from these considerations, the question remains of whether inheritance will be incorporated in the distributed philosophy, as the concept seems 'central.'[16] It certainly conflicts with the idea of fully open systems, which rely heavily on the notion of object (see [Bendrame 88]).

4.3.4 Location and migration of objects

An issue strongly connected with the previous two is that of object location and migration. It concerns the capabilities a language offers for hiding, exhibiting and manipulating the location of an object on the underlying machine (e.g. the network –virtual or real.) It is connected with location and migration transparencies.

location There are two aspects to location. The first is whether the language allows the visibility (and control) so that objects can be placed on specific loci under program control. The second is whether this visibility is permeated to other language features, i.e. if it is necessary to refer to the location of an object in situations other than placing or removing it from a location. For example, it is generally undesirable to have to specify an object's location to communicate with it.

migration It concerns whether it is possible to move an object from its present location to some other place, and whether there are any restrictions, e.g. that the object must not be currently executing. The same remark on visibility applies here. The only available example of fully mobile objects in a language is Emerald.

Note that both the features described above may also provided at a system –i.e. run-time– level, 'below' the language (e.g. the system might move objects to do automatic load-balancing). This is not discussed by the designers of the languages considered, except in Argus, where objects are migrated and allocated under different conditions, for it is done in the face of failures, and then the objects are 'restarted.'

[15] Of code or abstract interface [Wegner 87].

[16] It tends to create too many links –dependencies– among the components.

4.3.5 To what extent is the object boundary respected?

A problem solving scheme often used is to give the same problem to several different concurrent problem solvers. When one of them comes up with the solution, the remaining processes are aborted.

In distributed systems or databases, it is usual to start several concurrent instances of the same request to different servers (which are replicated to increase reliability and availability). The first in answering (that which is nearest or less loaded) is used (see the mailbox example in [Liskov 82]), the rest being aborted.

The preceding two situations have in common the fact that in certain concurrent environments one may start processes and then decide they are no longer useful before they perform their service, i.e., finish naturally. It is necessary to contemplate a mechanism for externally terminating them.[17]

In the case of servers that are active objects, however, the picture is not so simple, because a distinction must be made between the object that offers a service and the process that realizes it. Now the client of a service is no longer the owner of the computation that performs it; such computation belongs to the –active– object where it runs. This is in accordance with the ideas of autonomy and encapsulation. How is it possible to terminate a useless computation without violating the encapsulation of the enclosing object?

It is crucial to understand that the entity we deal with now (i.e. the entity with which the users interact) is an object, whose mean life is much longer than that of the services it performs (in particular it has a state that persists across instances of the service).[18] So it should be clear at this point that we want to get rid of the service but not of the object, that is, we are not interested in terminating (killing) the object (with the ensuing loss of its state).

One possible solution is to include an ad-hoc mechanism which punctures –in a 'controlled' way– the object's shell. This mechanism may be designed carefully, but this does not clear the reservations pointed out above.[19] There are interesting mechanisms (notably the very elegant one proposed in OTM [Hogg 87]) in which some conditions may be attached to a parallel block for determining successful completion before completion of all its components. In this case the termination is implicit (the runtime system takes care of it). Full extensions of these mechanisms are of dubious feasibility.

An alternative is proposed in ABCL/1, where an object may forced to leave its current activity (which may later be resumed or aborted) to start a higher priority one.[20] This is accomplished via an interrupt ("express") message. The object receives the information

[17] Of course, the termination must be 'clean' and the Terminator [Feuerstein 87] have the necessary permissions, i.e., must be acting on behalf of the client.

[18] Bear in mind that the service might be honored by creating an ad-hoc thread for each request, or via a single thread and a queue, but this is immaterial to our discussion, since we do not care for the state of the service thread after it has performed its duty.

[19] Roughly: what do we want object-orientedness for if we misuse it afterwards?

[20] This mechanism is low-level and was not intended just for this.

that a certain service it is performing is no longer needed, and can decide what to do about that. Notice how responsibilities are established: the requester of the service fulfilled its part by just making it known it no longer wants the service, thus freeing itself of any further commitment on it. It is the responsibility of the owner of the service (i.e. the active object that offers it) to deal with termination issues. (And of course the server may in turn rely on others to accomplish its task, so it becomes a requester asking for termination of related services, and it cascades on...) Note the cleaner separation of concerns and autonomy achieved in this model.[21] We believe this kind of solution to be better than the first both from an object oriented perspective (because the object boundary remains intact) and a distributed perspective (because the decisions on local entities may be suggested remotely but remain strictly local, i.e., autonomy is preserved).

4.4 Fault tolerance

One of the features that distinguish distributed from centralized systems is their behavior in the face of failures. In centralized systems the situation is usually binary: the machine (the system) may be either up (working normally) or down (crashed). Of course a peripheral may fail without compromising the rest of the system, but if the CPU fails, the whole system is down.

This is not the case in distributed systems. Being a set of loosely coupled computers, it may well happen that some are crashed while others keep on working perfectly well. For several reasons well discussed elsewhere (see [Lijtmaer 88, Ancilotti 88, Wulf 81]), it is strongly desirable that the system as a whole not come down as a result of the failure of one of its components. The techniques that deal with these issues are known as fault tolerance.

4.4.1 atomicity

Such techniques rely in several powerful mechanisms, notably that of atomic actions (transactions in database literature, actions in Argus jargon). Very roughly (see [Moss 85] for a detailed account), the idea is to perform a computation passing certain milestones ('checkpoints') that mark points of recovery in the face of failures. If a failure occurs, the system restores the state to the last checkpoint (i.e., does a 'rollback') and starts again from there. The key idea is that the system must guarantee 'all-or-nothing' semantics, i.e. either the operation completes successfully, or things are left as if the operation had never happened. Considering that concurrency is involved, the situation is complicated by the fact that some threads may be executing elsewhere on behalf of an aborted operation. Such 'orphaned' activities must be detected and terminated.

In connection with aborting computations in active objects, it was stressed in the section on encapsulation that a solution which respects the object boundary is desirable. This is

[21] These ideas are not new: see the concept of manager of a resource in operating systems.

not always the case, however, when one deals with fault tolerance in a distributed system. In rollback situations the system must be able to guarantee that some processes (threads) will be aborted, regardless of local idiosyncrasies. That is the case of mechanisms such as nested transactions [Moss 85].

So we have identified another important language feature: whether it offers mechanisms for executing atomic actions. Argus [Liskov 82, 88] does.

4.5 Transparencies

The notion of *transparency* is probably as old as the programming of computers. It comes from the desire to be able to 'forget', abstract, or hide certain underlying details, thus easing the managing of the whole. The word has been used in a range of situations. In operating systems [Peterson 85], for instance, the term is used to denote only what we call *access transparency*: a system is transparent if a user accesses all the files in the same way regardless of where they reside. We are interested in defining a more general idea, such that the preceding be one possible instance.

We say that a language (or system) is *transparent with respect to one of its features* if the details of that feature are hidden at the language level. In other words, if one need not care about managing such aspects. For example, a system may have *evolution transparency* if it is possible to allow change parts of it without disturbing the rest. (Consider replacing a module for a new one that implements some new functionalities preserving the old ones. This is possible in Argus where a 'guardian' offering a like interface may be inserted dynamically in an application, with the runtime system silently taking care of redoing bindings, etc., see [Liskov 83].) We adapt the definition of transparency and the relevant transparencies to our needs. A more precise definition requires a framework that exceeds the scope of this work, but can be found in [Bendrame 88], along with the definitions that correspond to the exhaustive list of transparencies given in [Lijtmaer 88] in connection with distributed systems.

So the **transparencies** (adapted from [Bendrame 88]) are described below. A language or system is transparent with respect to:

location if any operation on the entities of the language can be performed without knowledge of where the entity is located.

concurrency if a consistent view of an object is guaranteed even under its concurrent use.

fault if failures in the system do not compromise the execution of the application. The ideal expectation is that the behavior of the system is unaltered; in reality this is replaced by recovery mechanisms with reasonable semantics in the face of failures.

migration if displacing the entities about the system does not affect the applications in which these entities participate.

evolution if requirements (performance or functionality) may be changed without having to redesign the system (see example above)

It is noteworthy that the presence of a transparency is not intrinsically good. Sometimes a transparency will be desirable, but sometimes it will hide some characteristic that needs to be manipulated. And it may conflict with other transparencies, sometimes preventing them altogether.

a remark A final point before leaving the transparencies is their relation to isolated features as presented so far in this section. The reader will have noticed by now the important intersection between those features and (the names of) the transparencies given. Transparencies provide an additional vantage point to look at the interaction of concurrency and object-orientedness.

5 Conclusions and further work

The main conclusion is one that somehow prevents other conclusions. It is the recognition that the area we are trying to probe is very new and so still wide open and uncharted. This view is shared by others ([Moss 88], [Hogg 88], [Nierstrasz 88]). This is the reason for the lack of uniformity in concepts and terminology.

Another conclusion is that it is essential to formulate unifying models; at least we should strive to agree on what are the basic concepts that matter here. We believe our work is a step in that direction as it helps identify important issues.

An intuition that was present at the beginning –and that partially prompted this research– emerges stronger after the work: the marriage of the object model and concurrency is very interesting in at least two counts. One, the promises it holds in terms of easing the task of designing and building concurrent systems. And the other –last but not at all least– is the challenge of making it possible, conceiving and really building the abstract models that allow a deeper understanding and the ulterior formalization of the features pointed out in this work.

So, the road to follow is clear: to devise a model that adds concurrency to a more precise object model. In the short term, the first step is to isolate a set of elemental mechanisms (in an axiomatic sense) able to embody both the notions of concurrency and the object model.

Acknowledgments

Since this is (hopefully) the last document I will write as a student at ESLAI, I must express my gratitude to all who shared the experience, contributing to make ESLAI a very pleasant place to stay and work, despite occasional rough times.

I am also very grateful to the following people:

Norma Lijtmaer for her support and guidance and for giving me all kinds of transparencies. Carlo Ghezzi was my maestro of programming languages, and oriented my first attempts to define this work. (As usual, the shortcomings and errors in my work remain entirely mine.)

Carina Bendrame furnished some references and pointers, read and criticized parts of this work, and kept the coffee flowing during endless nights in our shared office. She graciously shared her findings from her own research on transparencies.

Marcelo Fiore asked pointed questions that led to some good insights. Luis Marrone and Claudio Hermida helped with the text formatting. ESLAI provided e-mail services that let me discuss some of the points with authorities in the field. ESLAI's second floor staff, especially Feliciano Argüello and Sonia Cairoli, provided essential logistic support.

Paolo Ancilotti introduced me to the subtleties and beauty of concurrent programming, and Ugo Montanari made me look at it all from a different point of view.

Vangelis and Vienna Philharmonic Orchestra supplied the music.

The far-away friends:

Eliot Moss (University of Massachusets at Amherst) made some (encouraging) remarks on OOC and shared doubts with me. Oscar Nierstrasz (Université de Genéve) cared to send a whole paper replying to my questions on OOC. John Hogg (University of Toronto): shared ideas and doubts on threads. Mehdi Jazayeri (Hewlett Packard Labs, Palo Alto) offered some pointers on threads/lightweight processes.

And last, but not least, to those who could not give technical or logistical help, but stayed there all the time, and believed it was possible: Cristina Juárez, Nicolás Pedregal and Sibila Martin.

References

[Agha 84] Gul Agha. Semantic Considerations in the Actor Paradigm of Concurrent Computation. In: Lecture Notes in Computer Science 197, Springer 1984.

[Ancilotti 88] Paolo Ancilotti and Maurelio Boari. Programmazione Concorrente.

[Barry 87] Brian M. Barry, John R. Altorry, John R. Altoft, D. A. Thomas, Mike Wilson. Using Objects to Build Radar ESM Systems. In: OOPSLA 87. The

[Bendrame 88] Carina Bendrame. Trabajo de Grado. ESLAI, December 1988.

[Black 86] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy. Object Structure in the Emerald System. Proceedings OOPSLA 86.

[Brinch Hansen 78] Per Brinch Hansen. The programming language Concurrent Pascal. In: [Gries 78]

[Broy 81] Manfred Broy. On Language Constructs for Concurrent Programs. Lecture Notes in Computer Science 111, 1981. Compares a sample of classical constructs studying their algebraic properties. It also considers applicative constructs (such as data flow languages).

[Cajías 88] Isabel Cajías, Esteban Feuerstein, Cristóbal Pedregal, Nora Szasz. Implementación de un Runtime Support para occam. Informe interno ESLAI (unpublished).

[Cardelli 85] Luca Cardelli and Peter Wegner. On understanding Types, Data Abstraction and Polymorphism. ACM Comp. Surveys 17(4) December 1985.

[Cox 86] B. J. Cox. Object Oriented Programming – An Evolutionary Approach. Addison-Wesley, Reading, Mass., 1986.

[Dahl 72] Ole-Johan Dahl and C. A. R. Hoare. Hierarchical Program Structures. in: O-J Dahl, E. W. Dijkstra, C. A. R. Hoare: Structured Programming, Academic Press, 1972.

[Doeppner 87a] Thomas. W. Doeppner. A Threads Tutorial. Brown University Technical Report CS-87-06, March 30, 1987. Threads programmer's manual.See [Doeppner 87b]

[Doeppner 87b] Thomas. W. Doeppner. Threads: A System for the Support of Concurrent Programming. Brown University Technical Report CS-87-11, June 16, 1987. Introduces the concept of concurrent threads (claimed to be "more abstract" than Mach's, see [Rashid 88]), discusses how the abstraction is built in layers.

[Feuerstein 87] Esteban Z. Feuerstein and Daniel N. Yankelevich. Personal communication. ESLAI, 1987.

[Fitzgerald 86] Robert Fitzgerald and Richard F. Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. ACM TOCS 147-177 4(2), May 1986

[Ghezzi 87] Carlo Ghezzi and Mehdi Jazayeri. Programming Language Concepts. 2nd ed., John Wiley. 1987.

[Goguen 86] Joseph Goguen and José Meseguer. Foundations and Extensions of Object-Oriented Programming. SIGPLAN 21(10), October 1986. Exposes algebraic foundations for object-oriented programming and its integration with functional programming. Proposes an operational semantics through rewrite rules.

[Goldberg 83] Adele Goldberg and D. Robson. Smalltalk 80: The Language and its Implementation. Addison-Wesley, May 1983. The book on Smalltalk by its designers and implementors.

[Gries 78] David Gries. Programming Methodology: A Collection of Articles by Members of IFIP WG2.3. Springer, 1978.

[Guttag 78] John V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. In: [Gries 78]

[Hewitt 77] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, 8(3) pp 323-364, June 1977 Seminal paper on Actors.

[Hewitt 84] Carl Hewitt, Tom Reinhardt, Gul Agha, Giuseppe Attardi. Linguistic Support of Receptionists for Shared Resources. In: Lecture Notes in Computer Science 197, Springer 1984. Interesting because discusses actors and concurrency in more detail.

[Hoare 78] Carl Anthony Richard Hoare. Towards a theory of parallel programming. In: [Gries 78]

[Hoare 85] Carl Anthony Richard Hoare. Communicating Sequential Processes. Prentice-Hall 1985.

[Hogg 87] John Hogg and Steven Weiser. OTM: Applying Objects to Tasks. Proceedings of OOPSLA 87. Describes an office automation programming language designed in the object-based paradigm, which offers very interesting constructs for dealing with concurrency.

[Hogg 88] John Hogg. Personal Communication. November 1988.

[Huberman 88] Bernardo A. Huberman (ed.) The Ecology of Computation. Elsevier Science Publishers B.V. (North-Holland), 1988.

[Jazayeri 88] Mehdi Jazayeri. Personal Communication. October 1988.

[Jones 86] Michael B. Jones and Richard F Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. OOPSLA 86.

[Jul 87] Eric Jul, Henry Levy, Norman Hutchinson, Andrew Black. Fine-Grained Mobility in the Emerald System. Proceedings OOPSLA 87. Focuses in the ability to migrate objects (even while active) across nodes of a distributed system.

[Kahn 88] Kenneth M. Kahn and Mark S. Miller. Language Design and Open Systems. In: [Huberman 88] Dicusses some language features related to encapsulation, as seen from the requirements of open systems.

[Knuth 69] Donald E. Knuth. The Art of Computer Programming. Addison-Wesley, 1969.

[Kung 88] H. T. Kung. Parallel Processing. In VI Escola de Computa cao, Campinas, SP, Brazil, July 1988.

[Lampson 80] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. Comm. ACM 23(2), February 1980. One of the first papers to introduce the notion of "lightweight process."

[Lijtmaer 88] Norma Lijtmaer. Lecture notes of the course "Estructura e Infraestructura de Sistemas Distribuidos". ESLAI, November 1988.

[Liskov 74] Barbara Liskov, Stephen Zilles. Programming with abstract data types. Proc. ACM SIGPLAN Conf. Very High Level Languages, SIGPLAN Notices 9, April 1974.

[Liskov 82] Barbara Liskov. The Argus Language and System. In Lecture Notes in Computer Science 190, Springer 1982. Overall description of the distributed language and system Argus. See also [Liskov 88]

[Liskov 83] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM Trans. on Prog. Lang. and Sys. 5(3) July 1983.

[Liskov 86] Barbara Liskov and John Guttag. Abstraction and Specification in Program Development. MIT Press, 1986.

[Liskov 87] Barbara Liskov, Dorothy Curtis, Paul Johnson, Robert Scheifler. Implementation of Argus. Op Sys Review 21(5) 1987. Special Issue Proceedings of Eleventh ACM Symposium on Operating System Principles. Describes the implementation of Argus on top of unix. Of interest is the implementation of "threads" (called simply "processes" in [Liskov 82])

[Liskov 88] Barbara Liskov. Distributed Programming in Argus. Comm ACM 31(3), March 1988. Updated and shorter version of [Liskov 82], explains Argus through an example. Best Argus primer.

[LNCS 197] S. D. Brookes, A. W. Roscoe, G. Winskel (eds.). Seminar on Concurrency. Carnegie-Mellon University, July 9–11, 1984. Lecture Notes in Computer Science 197, Springer Verlag, 1984

[McJones 87] Paul R. McJones and Garret F. Swart. Evolving the UNIX System Interface to Support Multithreaded Programs. DEC Systems Research Center, Report 21, September 1987. Describes efforts at DEC-SRC to extend unix to add internal concurrency to its processes with minimal impact on the interface.

[Meyer 88] Bertrand Meyer. Object-Oriented Software Construction. Prentice-Hall, 1988. Introduces the object-oriented language Eiffel with a software engineering viewpoint. Particularly valuable are the discussions about the need for certain features (such as multiple inheritance) in an o-o language.

[Montanari 88] Ugo Montanari. Several personal communications ("Sulla natura de l'Informática e perché io lavoro en questa disciplina"). ESLAI, 1988.

[Moss 85] J. Eliot B. Moss. Nested Transactions: an approach to reliable distributed computing. The MIT Press 1985.

[Moss 88] Eliot Moss. Personal Communication. November 1988.

[Nelson 81] Bruce J. Nelson. Remote Procedure Call. PhD Dissertation, Carnegie-Mellon University. Internal Report CMU-CS-81-119. Reprinted as XEROX PARC Report CSL-81-9, 1981.

[Nierstrasz 87] Oscar M. Nierstrasz. Active Objects in Hybrid. Proceedings OOPSLA 87.

[Nierstrasz 88] Oscar M. Nierstrasz. Two Models of Concurrent Objects. Position Paper presented at the workshop on Object-Oriented Concurrency OOPSLA 88.

[OOPSLA 86] Object-Oriented Programming Systems, Languages and Applications. Conference Proceedings, Orlando, October 4-8, 1987.

[OOPSLA 87] Object-Oriented Programming Systems, Languages and Applications. Conference Proceedings, 1987

[Peterson 85] James L. Peterson and Abraham Silberschatz. Operating Systems Concepts (2 ed.). Addison-Wesley, 1985.

[Power 88] Leigh Power (reporter). Panel-Discussion: Object-Oriented Concurrency. addendum to OOPSLA 87, May 1988.

[Rashid 88] Richard F. Rashid. From RIG to Accent to Mach: The Evolution of a Network Operating System. In: [Huberman 88].

[Stein 87] Lynn Stein. Delegation is inheritance. In [OOPSLA 87]

[Stefik 84] Mark Stefik and Daniel G. Bobrow. Object-Oriented Programming: Themes and Variations. The AI Magazine, 1984.

[Strom 84] Robert E. Strom and Shaula Yemini. The NIL Distributed Systems Programming Language: A Status Report. In: Lecture Notes in Computer Science 197, Springer 1984.

[Wegner 87] Peter Wegner. Dimensions of Object-Based Language Design. OOPSLA 1987.

[Wirth 82] Niklaus With. Modula-2 (3rd ed.) New York, NY: Springer Verlag, 1982.

[Takahashi 88] Tadao Takahashi. Introdućao a Programaćao Orientada a Objetos. III EBAI. Curitiba, PR, Brazil, January 1988.

[Wulf 81] William A. Wulf, Roy Levin, Samuel P. Harbison. Hydra/C.mmp: An Experimental Computer System. McGraw-Hill, 1981.

[Yokote 86] Yasuhito Yokote and Mario Tokoro. The Design and Implementation of Concurrent Smalltalk. Proceedings of OOPSLA 86.

[Yokote 87] Yasuhito Yokote and Mario Tokoro. Experience and Evolution of ConcurrentSmalltalk. Proceedings of OOPSLA 87.

[Yonezawa 86] Akinori Yonezawa, Jean-Pierre Briot, Etsuya Shibayama. Object-Oriented concurrent Programming in ABCL/1. In Proceedings of OOPSLA 86.