JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 4
Special issue: TOOLS USA 2003

Concurrent Object-Oriented Programming: The MP-EIFFEL Approach

Miguel Oliveira e Silva, DET–IEETA, University of Aveiro, Portugal

This article evaluates several possible approaches for integrating concurrency into object-oriented programming languages, presenting afterwards, a new language named MP-EIFFEL. MP-EIFFEL was designed attempting to include all the essential properties of both concurrent and object-oriented programming with simplicity and safety. A special care was taken to achieve the orthogonality of all the language mechanisms, allowing their joint use without unsafe side-effects (such as inheritance anomalies).

1 INTRODUCTION

In this article a new concurrent object-oriented programming language is presented. This language – named MP-EIFFEL (Multi-Processor EIFFEL) – is developed as an extension to the sequential object-oriented language EIFFEL. This choice is not accidental. EIFFEL's powerful, safe, and simple object-oriented semantics proved to be an excellent framework to introduce concurrency mechanisms.

The choices made for the language concurrent mechanisms, and their semantics, resulted from an attempt to provide a safe and simple integration of the essential properties of both worlds: concurrent and sequential object-oriented programming.

In section 2 the essential characteristics, requirements, problems, and elementary solutions of concurrent programming are briefly reviewed. Object-oriented sequential programming is then briefly introduced, giving emphasis to its fundamental and useful properties.

Since this work is about creating programming language mechanisms, some important language design rules are enumerated and succinctly justified. Those rules will be used to justify the choices made in MP-EIFFEL.

Several existing possibilities for integrating concurrency in an object-oriented language are presented and evaluated in section 4. A special care is taken to study the conditions under which unsafe interference may arise from the joint use of some mechanisms, such as the well known problem of "inheritance anomalies". It is shown that such problems can be prevented without compromising neither of the essential properties of both worlds.

Other important concurrency safety problems such as deadlocks, and other synchronization requirements such as scheduling are briefly approached.

Finally, the MP-EIFFEL language is presented.

Cite this article as follows: Miguel Oliveira e Silva: "Concurrent Object-Oriented Programming: The MP-EIFFEL Approach", in *Journal of Object Technology*, vol. 3, no. 4, April 2004, Special issue: TOOLS USA 2003, pages 97–124, http://www.jot.fm/issues/issue_2004_04/article6

MP-Eiffel aims to be a safe concurrent language, where unsynchronized accesses to shared resources are statically prevented (unlike, for example, Java).

Concurrent activities are performed by abstract processors (a term reused from Meyer's approach to concurrency: SCOOP), which may access shared objects, and also communicate directly with each other through a mechanism named "trigger".

Concurrent access to objects use a readers-writer semantics, in which side-effect free accesses may occur simultaneously (the only acceptable form of intra-object concurrency), but at most only one writer is allowed to proceed.

The type system is used to safely control the access to shared objects, with the introduction of two type modifiers: shared and remote.

SCOOP semantics of reusing the separate part of preconditions for conditional synchronization is also used in MP-Eiffel.

The exception handling mechanism is adapted for a concurrent environment in which a synchronous communication mechanism (either when accessing a shared object, or when sending a synchronous message to another processor) retains Eiffel's normal semantics. For asynchronous communication between processors the message communication status is verifiable by special direct inter-processor communication functions (synchronous).

MP-Eiffel takes the view of removing from programs all time dependent requirements (scheduling, exclusion, etc.) which do not affect their logical semantics, delegating them either to the language semantics (exclusion, deadlocks) or to a separate description program (scheduling, real-time requirements). This way the programs are safer and easier to understand.

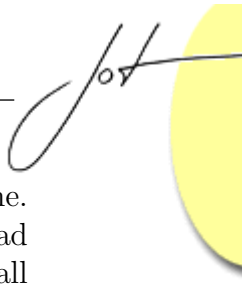
2 CONCURRENT PROGRAMMING

In this section we will make a synthesis of the most important characteristics of a concurrent programming system (programming language and libraries). A special care will be taken to generalize as much as possible the concepts involved, and to identify the most serious problems raised by this type of programming.

Abstract concurrent programming

Since we are interested in the essential properties of concurrency, we don't want to bind our concurrent processing entities to specific "lower level" realizations such as threads, processes, object request brokers, MPI, PVM, or any other. Instead, we will use an abstract notion of "processor" adapted from Meyer [Mey97, page 964]:

> A processor is an autonomous thread of control capable of supporting the sequential execution of instructions.



Nothing is assumed on how this virtual processor will be implemented at runtime. We won't even exclude the possibility that a processor may start by being a thread in one machine and end as a process in another through a remote procedure call mechanism.

In general, concurrency does not require simultaneous (parallel) processor execution. A concurrent program can be thought as the concurrent execution of sequential "programs", one for each processor (of course, they may depend highly on each other).

From a concurrent programming system we expect the ability to – safely – specify and control two things:

- concurrent execution of processors;
- inter-processor synchronization and communication.

Concurrent execution of processors

Concurrent systems need mechanisms to start, support, and terminate the execution of processors, either as part of the programming language constructs or provided in a library. In particular they need to assign processors to specific processing devices (threads, processes, computers in a distributed system, or any other). This assignment will be called heterogeneous if the concurrent system allows the assignment of different processing devices for processors, otherwise it will be classified as homogeneous processor mapping.

Heterogeneous processor mapping is a desirable property since it enforces the decoupling between a concurrent program and specific processing devices. However, specific types of concurrent programming, such as real-time and embedded systems may pose strong restrictions to valid (feasible) processor mappings.

Inter-processor synchronization and communication

To ensure a correct inter-processor interaction it is necessary to enforce proper timing constraints between them. Such synchronization requirements may arise from the necessity to guarantee safety properties [Lam83], the verification of some condition, the need to impose a specific processor scheduling strategy [RF77], or several of these possibilities at the same time.

Safety

A safety problem exists when a program may have an unpredictable (erroneous) behavior due to incorrect inter-processor synchronization.

This type of errors is the most serious correctness problem raised by concurrent programming. They are dependent on low level timing conditions which might be hard to reproduce and detect.

Race conditions are the simplest of this type of problems. They exist whenever several processors simultaneously attempt to modify a shared resource without a correct exclusion synchronization (which may prevent any of them to do what they expect, leaving the resource in an inconsistent unsafe state). This problem is solved by protecting the access to the mutable shared resource within a critical section, using for example, semaphores [Dij68].

One advantage of a concurrent programming language when compared with a library approach for concurrency (such as POSIX threads in C [But97]), is that it gives the possibility of prevention of this type of errors using the language semantics, as will be shown in the next section.

Other safety problems created by synchronization mechanisms may also exist, such as deadlocks, and the more general problem of ensuring liveness [Lam83].

For correctness and robustness reasons, our goal should be to ensure that no such errors would ever occur in a concurrent program.

Conditional synchronization

Frequently the access to a critical region of a shared resource depends not only on exclusion needs, but also on the verification of some resource related condition. For example, a printing processor is required to conditionally wait for the non-emptiness of its shared job queue.

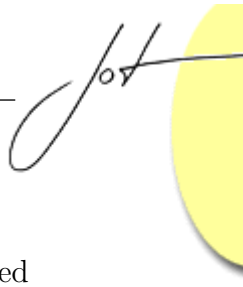
There are several mechanisms well adapted to this problem such as conditional critical regions and monitors [Hoa74].

Scheduling

Scheduling is the strategy for choosing the processor to be executed when processor contention occurs (which may depend on limited processing devices resources, or resulting from an attempt to access a shared resource).

In general, scheduling depends on three factors [RF77]: the decision mode, the priority function, and an arbitration rule. The decision mode characterizes the time instances in which processor scheduling is computed, the priority function is the processor ordering algorithm, and the arbitration rule is the strategy used to choose between equal priority processors.

The scheduling choice may affect the safety of concurrent programs, since it may prevent some deadlock problems, or – such as when an extremely unfair algorithm is used – it may pose other liveness problems such as starvation to some processor.



Deadlocks

Deadlocks are situations in which processors wait forever for each others allocated resources. In order for a deadlock to occur, four necessary conditions must be verified [CES71]:

1. mutual exclusion (exclusive access to a resource);
2. hold and wait (waiting for resources while holding at least one);
3. no preemption;
4. circular wait.

The absence of any of these conditions is sufficient to prevent deadlocks.

Three possible approaches can be applied to handle this problem [CES71]:

1. Prevention;
2. Avoidance;
3. Detection.

Deadlock prevention ensures the absence of deadlocks by statically assuring that at least one of the four conditions does not apply. For example, if a processor is only allowed to hold a unique resource (preemption allowed), or if processors must acquire all resources at the same time (hold and wait denied), or if an ordered allocation of resources is imposed (circular waiting impossible), then deadlocks cannot occur. However, some care must be taken in the usage of prevention techniques since they tend to be too costly to the overall performance of the concurrent system.

Another safe possibility is to use deadlock avoidance techniques. If information about current and future resource allocation is available, then that knowledge can be used to avoid circular waits (see, for example, the Banker's algorithm [Dij68, Hab69]).

The third possibility is to have deadlock detection algorithms and recovery strategies (which may use exception handling techniques).

Only the first two approaches are guaranteed to be safe – since they do not affect the normal execution of a processor – so those are the ones a safe language should consider.

Unlike the mutual exclusion problem – which is a local problem with a local solution – this problem arises due to a global (system wide) interference between processors. This characteristic makes it a much more difficult problem to tackle.

Communication

By far the most important request for synchronization is the necessity for inter-processor communication. A communication mechanism is said to be synchronous in relation to a processor if that processor may be required to wait (block), otherwise it is called asynchronous.

There are two basic models for inter-processor communication:

- message passing (direct);
- shared memory[1] (indirect).

In message passing, processors communicate directly using some sort of a (real or virtual) point-to-point connection. This form of communication is well adapted for loosely connected processors (as in a distributed system), and for client-server topologies. The sender processor in this communication abstraction can either be synchronous or asynchronous.

Shared memory is an indirect inter-processor mechanism, in which processors communicate by using a shared writable and readable entity. It is well adapted for processors who need to frequently share mutable information. Shared memory requires a synchronous communication.

As noted in [LN78] either model can be converted into the other, so one could argue that in principle a concurrent language only needs one of them. However, they represent different processor communication abstractions and have different requirements to the underlying execution system, so it is also defensible to have them both.

3 DESIGN OF OBJECT-ORIENTED LANGUAGES

Meyer [Mey97, page 147] defines object-oriented software construction as:

> (...) the building of software systems as structured collections of possibly partial abstract data type implementations.

An Abstract Data Type (ADT) [LZ74, Mey97] is, by definition, a type that is completely defined by the external operations it provides, and its semantics (axioms and preconditions). In object-oriented programming ADTs are implemented with classes [Mey97], and their run-time instances are called objects.

Making the software's topmost structure depend only on ADTs and their relations, simplifies the development of complex systems, due to its modularity and understandability [Mey97, page 39].

[1] The term "shared memory" is used throughout this article to mean a unique uniform addressing system usable by all processors.

An object-oriented programming language is characterized by some fundamental properties:

- Objects are first-class entities, aware of their behavior (defined by its class), to whom some operations may be requested, and which are themselves usable as arguments on operation requests;
- There is an explicit and uniform communication mechanism between objects (message passing in SmallTalk terminology, or feature calling in Eiffel, C++ or Java);
- The ability to transparently use objects on behalf of any type applicable to its class (inclusion polymorphism).
- The ability for a class to inherit from others, reusing or redefining their features.

Additionally some languages also provide other important mechanisms:

- Information hiding [Par72b, Par72a], through explicit private and public class interfaces[2];
- The ability to specify the ADT semantics (invariants and preconditions) in the class, promoting Design by Contract [Mey97, chapter 11];
- Multiple inheritance;
- Parametric polymorphism [CW85][Mey97, chapter 10];
- Exception handling;
- Class attribute protection [Mey97, page 206] (an attribute can only be changed by routines of its class).

Some authors [Car88, CHC90] argue for the necessity to separate the notion of inclusion polymorphism (subtyping), from the ability to reuse features using inheritance (subclassing). In the remaining of this article, following the language design rules of simplicity and uniqueness presented ahead, we will follow Meyer's [Mey97] approach in which the same language mechanism – inheritance – provides both possibilities (at the same time if needed). Although simpler, this choice poses an extra burden on the language type system to guarantee the detection of all type errors during program compilation.

Hereafter we shall use the expression "object entity", to designate the programming language constructs which handles object references or values (in Eiffel they

[2] Eiffel goes beyond this limited distinction allowing a much more tunable export control.

are class attributes, routines local variables or their formal arguments [Mey97, page 213]).

The challenge is how to integrate concurrent and object-oriented systems without compromising none of their essential properties.

Language design guidelines

To ease the process of construction of high quality programs (especially in what concerns correctness), it is not enough for a programming language to have all or most of desired mechanisms; the way they are put together and their usability are also essential.

On language design Hoare [Hoa73] notes that programming languages should help the programmer in the most difficult aspects of programming: program design, documentation and debugging.

The first fundamental rule presented by Hoare for good language design is simplicity, which can be stated as follows.

Rule 1 (Simplicity) *A programming language should be simple to understand and use.*

Of course a programming language is a tool to express solutions of problems, so although simplicity is essential on the implementation of its fundamental properties and mechanisms it is not a reason to ignore them. Otherwise an elusively simpler structure in the programming language may cause a much higher complexity in the programs. This leads to another essential rule: completeness.

Rule 2 (Completeness) *A programming language should implement the mechanisms, and have the properties considered essential to its methodology of problem solving.*

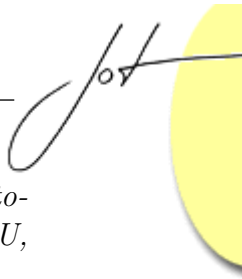
The second essential rule presented by Hoare regards safety[3] on language constructs usage.

Rule 3 (Safety) *Programming language mechanisms should be prevented from producing meaningless results.*

This rule justifies, for example, the use of a static type system – which prevents the occurrence of type errors at run time – and it will also be essential for evaluating the proposed concurrent mechanisms.

Other less essential rules should be taken into consideration when designing programming languages. That is the case of efficiency, which promotes the language use in resource demanding applications.

[3]Hoare uses the term "security" for this purpose.



Rule 4 (Efficiency) *The programming language should not compromise the automatic generation of programs which make an efficient use of system resources (CPU, memory, etc.).*

Meyer [Mey92, appendix B], presents two other rules.

Rule 5 (Uniqueness) *The language design should provide one good way to express every operation of interest; it should avoid providing two.*

Rule 6 (Consistency) *The language design should never depart from a small number of powerful ideas, taking them to their full realization.*

Other guidelines can be formulated which go along with these rules.

Rule 7 (Abstraction) *Programming language constructs should be completely defined using only their external abstract behavior, regardless of implementation.*

This rule promotes simplicity. The external abstract behavior of programming language constructs should be made simple, regardless of their possible implementations. Simple mechanisms are sometimes hard to implement correctly. For example, the ability to covariantly redefine feature signatures in Eiffel (which is a simple and useful behavior), is difficult to implement in order to ensure a safe system [Coo89][Mey97, page 621].

Rule 8 (Orthogonality) *Programming language constructs should be made as orthogonal as possible, in order to make them work correctly regardless of being used together with others.*

The observance of this rule simplifies the language semantics, and is a guarantee that no undesirable side-effect will occur when its mechanisms are jointly used. It is important to note that orthogonality applies to the constructs essential semantics, not to their eventual implementations. The implementation of a language mechanism may depend heavily on the context in which it is used. For example, the uniform inter-object communication mechanism (feature call or message passing using Eiffel or SmallTalk terminology) can be reused in a concurrent system for either of the two types of processor communication: shared memory or message passing.

4 THE ROAD TO OBJECT-ORIENTED CONCURRENCY

In this section we shall evaluate some object-oriented approaches for concurrency, in order to justify the choices adopted in our approach presented in the next section.

Processors and objects

There are several possibilities for integrating processors with objects. Some approaches promote processors to classes (//Eiffel [Car93], POOL [Ame87]), in which there is a special main feature with the processor's body and synchronization code.

This possibility raises many problems. It assumes that processors are valid ADTs, which is in general difficult to accept (it would be an ADT with a unique operation). If that concept is applied to the special concurrent case of a sequential program (uni-processor), then we would be at odds with the basic object-oriented design definition of constructing software systems as organized collections of ADTs implementations.

A better possibility is presented by the actors model [Agh86, AK99], in which instead of promoting processors to classes, classes are augmented with a processor, capable of processing and responding to a call to any of the class's features. This model is well adapted to the distributed modular message based nature of object-oriented programming, but has the drawback of making processors and classes hard-wired entities.

Meyer's proposal for concurrency with Eiffel [Mey97, page 951] – SCOOP[4] – goes much further by allowing a processor to handle multiple communicable objects (of different types if necessary). In SCOOP the concurrent nature (separateness) of an object may not be a property of its class, but simply a property of the object's entities which use it. This makes objects and processors more independent entities, simplifying the reuse of normal sequential classes in a concurrent system.

However, these two approaches – Actors and SCOOP – restrict our concurrency model to explicit message passing communication between processors (no object is ever executed by more than one processor), which although being adaptable to any concurrency need, may not be the more appropriate and efficient option. Frequently our concurrency needs demand direct resource sharing between processors in which the participation of a third processor is not the simplest and more appropriate programming approach (using a real life analogy: why restrict ourselves to use a unique "chauffeur" to handle a shared car, when anyone can drive).

A fourth option is to make objects and processors completely orthogonal entities, in which an object may be executed by different processors, and the processor creation depends on the use of some specific mechanism. This is the approach taken in several widely used concurrent systems such as Java threads and ADA's protected types. This possibility, however, if not done correctly creates more safety problems, resulting from concurrent access synchronization. Those problems will be studied below.

[4]Simple Concurrent Object-Oriented Programming

Intra-object concurrency

Intra-object concurrency is the possibility of more than one processor operating simultaneously inside an object. Is this type of concurrency acceptable? If so, in what conditions?

Since the fundamental theory behind object-oriented programming is taking classes as ADTs implementations, the answer to the first question is immediate: It can only be allowed if it does not compromise the class's ADT implementation.

In sequential programming an object is only usable during its stable times [Mey97, page 364]: after its creation, before or after terminating the execution of any of its features (times at which the class invariant must hold). When an object feature is being executed, the class invariant may not hold, in which case it could invalidate a feature call from another processor.

Also, a conditional wait for the invariant, and eventually the feature precondition, is not a sufficient condition to allow this type of concurrency, since it may create race conditions (the worst kind of concurrent errors), and it does not take into consideration the possibility of an incomplete invariant implementation (which is a common situation). A simpler sufficient (but not necessary) condition is to consider classes as monitors[5], thus ensuring mutual exclusion on the execution of any of its features. This is the approach followed, for example, by Actors, SCOOP and ADA's protected types.

JAVA completely ignores this problem, unacceptably leaving that responsibility to the programmer. For this reason JAVA is an unsafe language for concurrent programming [Han99].

One interesting exception to this mutual exclusion rule is to allow intra-object concurrency for simultaneous side-effect-free queries to objects. For this reason, a readers-writer model [CHP71] for object concurrent access is a more general preferable option (one writer excludes all other processors, but multiple readers can simultaneously access the object).

The enforcement of class attribute protection by the language (as happens with EIFFEL), is also a desirable property since it prevents the occurrence of a dangerous unsafe intra-object form of concurrency resulting from a change to an object state outside the control of its class.

Synchronizing processors

As seen in the previous section, it is necessary to impose appropriate timing constraints when processors compete for a shared resource (object, CPU). Such need may arise from safety reasons (mutual exclusion, liveness), conditional synchroniza-

[5] A curious situation since the importance of Simula's class concept was recognized by Hoare [Hoa74] and Hansen [Han93] when proposing monitors.

tion, or scheduling policies.

The integration of synchronization with object-oriented mechanisms has been one of the most difficult challenges, and has concentrated much of the investigation in this area.

The fundamental reason for this is simple to understand. One of the key abstractions that justifies much of the success of high-level sequential programming languages is the ability to ignore, during program development, timing constraints attached to the language constructs. A program expresses the causality between instructions, and ensures the verification of some logical, time independent, conditions before and after their execution [Flo67]. Concurrency may break this simple view of languages and programs. Object-oriented concurrency increases this problem due to the possible interference of some of its fundamental mechanisms such as inheritance.

The coexistence of concurrency with inheritance has been widely studied [Ame87, KL89, MY93]. The problems it raises were named "inheritance anomalies" by Matsuoka (et al.) [MY93].

Inheritance anomalies

An inheritance anomaly occurs when the synchronization scheme loses its initial goal (mutual exclusion, conditional access or scheduling) in a descendant class, forcing its explicit redefinition. In the presence of these problems, the use of inheritance may became unsafe, which is unacceptable for a concurrent object-oriented language.

Rule 9 (Inheritance Invariance) *A sufficient condition which prevents inheritance anomalies is the guarantee that the synchronization expected in any object entity in the program applies to all possible (conforming) objects it may be attached to.*

If the language synchronization code (portion of the program which specifies synchronization behavior) is explicitly part of the program (explicit synchronization), then the observance of this rule will most likely require the unsafe cooperation of the programmer. On the other hand, if implicit synchronization is used together with the type system in a static typed language, then there is the possibility for a static safety assurance. In order to reach that goal it is necessary to guarantee that the interaction with objects usable by more than one processor is done through a proper concurrency type modifier, and that the subtyping relation (conformance) imposes its observance.

Meyer's SCOOP approach uses this strategy by augmenting the type system with the modifier SEPARATE to denote entities that have references to objects belonging to other processors, and by statically ensuring the absence of "traitors": separate objects attached to a non-separate object entity [Mey97, page 973].

Synchronization programming

The preceding discussion suggests the existence of two apparently irreconcilable situations: Implicit synchronization disallows the possibility for imposing some specific synchronization need in the program, but on the other hand, when concurrency was presented in section 2 one of the requirements was to allow precisely that: the ability to specify (safe) synchronization schemes (such as a scheduling policy).

Implicit synchronization is coherent with the already mentioned idea that timing dependent constraints should be kept apart from language constructs in order to simplify their semantics and to allow its reuse on different (or even mutable) execution systems. Following this reasoning, including non-essential synchronization requirements in the program would be a form of overspecification.

A possible solution to this problem, following the orthogonality rule, is to have an external language (or configuration system) in which non-essential synchronization requirements may be expressed. The compiling and execution systems would have to validate such timing requirements (which may include real-time constraints), build and support the execution of programs in which the verification of those constraints is ensured (violations to them must be detectable and propagated to the program using an appropriate exception mechanism).

A similar approach is taken in [RA95] for real-time specifications.

This approach also provides a solution for implementing heterogeneous processor mapping. In his SCOOP approach, Meyer uses for this purpose a Concurrency Control File [Mey97, page 971].

Conditional synchronization

Although many synchronization constraints may, and most likely should, be separated from the program's logic, there are two important exceptions: exclusion (since it is a minimum requirement for a safe concurrent language); and conditional synchronization. Conditional synchronization depends highly on the program's state, which makes it difficult to separate it into an external safe specification.

How can then conditional synchronization be integrated in a concurrent language, without creating safety problems, and without interfering with the other language mechanisms?

In object-oriented languages, objects interact by calling each others public features (routines or attributes), and – as already noted – a safe concurrent interaction between objects must occur in the object's stable times. This restriction strongly suggests that conditional synchronization tests be located in the object's class interface (this also seems to be the simplest solution).

The next problem is to decide in which part of the class's interface it should be located. One possible scheme is to have a centralized conditional synchronization

control (such as behavior abstractions [KL89]) separated from the normal class interface. This option seems to be an acceptable choice for synchronization conditions that apply to the call of any public feature of the class, but it is highly questionable when they apply only to a subset of those features (or simply one of them). Also, this option is highly sensitive to partitioning of acceptable states inheritance anomalies [MY93]. A partial solution for this type of inheritance anomaly is to use method (routine) guards, in which conditional synchronization tests are attached to the routines requiring their observance. A complete solution requires that the guards also apply to eventual routine redefinitions in descendant classes.

Curiously (but not by chance) these requisites seem quite similar to those applicable to one of the semantic specification of routines: preconditions. This observation raises the question of how should assertions behave under a concurrent interaction? Assertions are correctness conditions, which must be verified in the places where they are declared. However, if an assertion (or part of it) depends on more than one processor (concurrent condition), its value may be, from time to time, false due to the (unpredictable) activity of another processor. This is a safety problem: one needs to lock the objects on which the assertion depends before the test is made, but then the assertion may sometimes fail due to unpredictable timing relations between competing processors, making it useless for correctness assurance (they became part of the problem, not the solution).

Again one is surprised with the similarity of such reasoning with the one which has led to the proposal of conditional critical regions and monitors [Hoa74]. Meyer, while studying this problem, called it the concurrent precondition paradox [Mey97, page 994].

Of course, one possible "solution" is to disallow the existence of assertions which depend on more than one processor. However, this would be a very poor solution, since it restricts tremendously the applicability of Design by Contract.

Another alternative is to remove the concurrent conditions from the assertions, and use them as conditional synchronization tests before assertion verification. This approach, though a safe one, implies the introduction of a new language construct and the duplication of the assertions semantics (it goes against the rules of simplicity, uniqueness and orthogonality), and, more importantly, it restricts the reuse of sequential classes under a concurrent environment (since they would lack conditional synchronization tests).

A much simpler solution is to make the assertion conditions that depend on more than one processor, conditional synchronization tests. This is Meyer's elegant solution to this problem [Mey97, page 996].

One last important observation on this topic. From all the assertions that are applicable to object's state times – invariants, preconditions and postconditions – this scheme only makes sense for preconditions. Invariants cannot be conditional synchronization tests, because they must always hold in the object's stable times regardless of its executing processors. Postconditions, on the other hand, apply

when the routine's work is already done, so there is nothing to wait for.

Deadlocks and other liveness problems

One last group of difficult synchronization problems remains to be studied in the context of its integration in an object-oriented language: deadlocks, livelocks, and other liveness problems. As mentioned previously, they should be approached as safety problems, and, as such, a safe concurrent language should be able to ensure its absence, either by prevention or avoidance techniques.

Not all of the four necessary conditions for deadlock resolution can safely be used for deadlock resolution under an object-oriented system. That is the case of "mutual exclusion" (if synchronous processor interaction is required) and "no preemption". Breaking either of these conditions would result in objects interacting while in unstable times. Therefore, the only two conditions that can be used to prevent or avoid deadlocks are the "hold and wait" and "circular wait" conditions.

Since deadlocks are a system problem (not the responsibility of any individual processor), their resolution requires system wide techniques. That is not a unique case in object-oriented languages: a similar situation occurs, for example, in the use of covariant feature signature redefinition in the presence of subtyping.

Again, a static typed language (with concurrent properties) with an implicit synchronization scheme seems to be a promising approach to the problem, due to the fact that it may statically gather information about probable resource allocation needs. A smart compiling system may use a mixture of prevention and avoidance techniques (depending on the resource allocation patterns) to ensure a safe program.

Also, these liveness problems will most likely be simplified if resource (object) allocation is concentrated and restricted to fewer places in the program.

At this time this is still an open problem requiring deeper analysis and experimentation.

Processors communication

Object-oriented programming uses a uniform inter-object communication language mechanism, so its reuse for processor communication arises naturally (simplicity, uniqueness, consistency, orthogonality and abstraction). If this adaptation is quite consensual, the same does not happen with the choice for the appropriate model: message passing or shared memory. At first sight, since objects communicate with each other through messages, the choice would seem obvious: message passing model. However, although both are message passing forms of communication, they apply to different things: objects and processors.

The message passing inter-processor model of communication would be the correct (and only) option if objects were at most handled by a unique processor (as

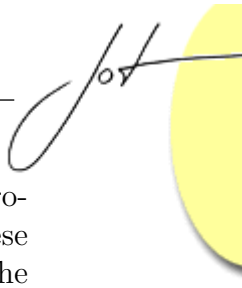
happens with Actor languages and SCOOP). However, if that is not the case, then that choice is not so clear. The question here is not whether objects communicate through message passing (which they always do), but which processor has the responsibility to fulfill the request executing the appropriate feature. If a concurrent program is seen as the aggregation of communicating sequential programs (one for each processor) using shared resources, then the simplest solution will be the shared memory model. As will be seen, from this choice results also a simpler and more efficient integration of the exception handling mechanism with concurrency. Nevertheless, as mentioned previously, the usefulness of a direct inter-processor communication mechanism, as exists in Actors or SCOOP, is obvious. It is useful, for example, to implement client-server program architectures, or for loosely connected distributed systems.

This is a situation where language design rules conflict with each other, and a choice must be made. Considering the uniqueness rule (rule 5) and the fact that either model can be converted into the other [LN78], the use of only one of them seems the correct approach. However, they represent two different ways of expressing concurrent programming, and so, without them, a language would be less complete. To convert one in the other, the resulting program loses simplicity (rule 2).

If a specific direct inter-processor (message passing) mechanism is to be devised, then a way must be found to identify processors (otherwise this form of communication would be senseless) and it is also necessary to study how this mechanism can be integrated within classes, since processors can only execute inside objects. Here we are confronted with an apparent paradoxical situation: in an approach which regards inter-object and inter-processor direct communication as different things, how can the latter be implemented within classes? After all, classes are object types, not processor types.

To solve this apparent paradox it is necessary to find answers to the processor identification problem. One possible approach would be to define a centralized processor registration, identification and communication mechanism, where processors could be named, and through which they could exchange messages. This possibility, however, has serious drawbacks:

- messages would most likely be restricted to data, to which an appropriate behavior could only be defined by the receiver processor (with no assurance that the data values are meaningful to it);
- it would be a mechanism outside the object-oriented program entities: classes;
- a communicating processor would have to rely on a third party entity for choosing the correct destination processor;
- it is a centralized scheme, which does not scale well to distributed systems, and may raise communication bottleneck problems;
- it is a rather complex solution.



All these problems – some of which might have some benefits in other programming environments – are not appropriate to object-oriented systems. In these systems, communication involves behavior (requested by the sender, and with the guarantee that it is meaningful to the receiver processor); classes should be the only topmost program structure; and software development should be decentralized.

On the other hand, if there was a simple way to bind processor identification to objects, then messages between processors could be requests for the execution of features on any of the receiver processor objects, which does not have the drawbacks of the presented centralized possibility. In fact, a simple solution to this problem already exists: SCOOP's approach is (almost) perfect for an object-oriented direct inter-processor communication mechanism. There, a processor is identified by the nonseparate objects it creates and it is responsible for handling any message (feature call) sent to them. To handle all messages without losses, it is necessary to attach a message queue to processors. Therefore, a message is handled when its turn has arrived and the processor is not doing anything else.

In this approach, if a procedure message is sent, then the communication will be naturally asynchronous (to the sender). On the other hand, a function (or an attribute) message will be synchronous (as will be seen, delaying the point of synchronization until the result is needed – as SCOOP proposes – might not be appropriate for a meaningful exception handling mechanism).

However, using this SCOOP approach for processor direct communication – in languages that separates both forms of communication (inter-object and inter-processor) – disregards the fact that, normally, classes are not built thinking about inter-processor direct communication. So, it is questionable that the interfaces (acceptable messages) to both worlds should be the same. Therefore, it makes sense to augment classes with an inter-processor export mechanism, in which the names of the communicable inter-processor features are listed.

Grabbing multiple objects

Frequently in concurrent programming it is necessary to ensure an exclusive access (or better yet: a readers-writer lock access mode) to other objects besides the current object for the execution of a code block. A possible approach to get this functionality is to add a new critical region instruction such as [Han72]:

```
region entities do
  ...
end
```

However, this option has the problem of moving object locking requests away from the current object stable times (which may pose extra difficulties for ensuring safety in deadlocks and other complex liveness problems). A simpler approach is

the one proposed in SCOOP, in which objects are held within the routine's body, and the criteria for choosing the objects to grab is simply the ones attached to the routine's concurrent formal arguments (separate in SCOOP). Also, with this approach – unlike the dedicated critical region instruction – it is easy to apply the Inheritance Invariance rule, making it immune to inheritance anomalies.

Another interesting positive side-effect of this choice is that – since it concentrates and restricts the places where locks are allowed – it will most likely simplify the resolution of deadlocks and other liveness problems. The example presented in the end of this article clearly illustrates this situation.

Concurrent exception handling

One last integration problem will be addressed: how to handle exceptions under a concurrent system.

Meyer [Mey97, page 412] defines exceptions as run-time events that may cause a routine to fail. A failure may arise from hardware related problems (lack of memory, or arithmetic operations faults), or, in general, resulting from a contract failure. In a disciplined exception handling mechanism (as exists in EIFFEL), a routine can either fail – reporting failure to its caller – or retry to fulfill the contract (failing again if unable to do that). Under a concurrent system new sources of failures may arise. Processors may fail, either by an unsolved exception, or, in a distributed system, due to a communication failure.

In a sequential language, the exception mechanism exploits the causal relation between the exception source (a program block) and its handler (declared within the scope of the exception). For example, in C++ the block is defined by TRY statements and exceptions handled in CATCH instructions. In EIFFEL every routine is a possible exception block, and exceptions are handled by RESCUE clauses, or propagated to the caller. This causality property is essential for a correct failure localization and its appropriate handling.

A similar scheme can be applied in a concurrent environment if this casual proximity relation persists. That is the case of a synchronous processor interaction (either with shared memory processor communication, or synchronous message passing), if no lazy evaluation (wait by necessity [Car89]) scheme is allowed (unlike SCOOP [Mey97, page 987]). With wait by necessity schemes, the required proximity relation between the caller and the callee may not hold, disallowing a correct exception handling.

When the callee object may involve more than one processor, two new problems arise: how can that object be used in a failure recovery (retry) attempt, and what will happen if, meanwhile, another processor attempts to use it?

One possibility for solving both problems would be to maintain the lock (exclusive access) to the failed routine's object until either there was a successful recovery

(from where everything would go on as if nothing had happen), or a processor failure occurred (which in a mechanism like EIFFEL's, occurs when the exception is propagated until the processor's creation routine is reached, and also fails). On processor failure, there could be an exception raised to any processor attempting to use that object. However, this is not a completely satisfactory solution, because the failure might have been the responsibility of the caller (not the callee), due, for example, to a precondition failure. Such event should not affect either the called routine's object, or other processors that might use it. A much better solution is to use this scheme only if the called routine's object is not in a stable state (in which case it is not usable anyway). Otherwise, the exception will only affect the caller, and a recovery attempt would be simply a new call request requiring appropriate synchronization. In a shared memory communication model this scheme is particularly simple and efficient since it does not require the collaboration of another processor. We will call this solution a disciplined synchronous exception handling.

Under asynchronous communication there is a different situation. In this case – by definition – no causal relation can be established between the failure and the message sender, which inhibits the possibility of using the proposed synchronous mechanism. Several approaches can be devised to handle this situation. If, in this form of communication, it is assumed that an eventual communication failure is not the responsibility of the sender, then a possible approach is simply not providing an asynchronous exception handling mechanism. However, this simpler choice (rule 1) is not complete (rule 2) because it would result on a higher complexity in programs requiring the confirmation of a successful communication. Another possibility is to use the asynchronous communication mechanism in the reverse direction generating – on communication failure – a special error notification message (that special feature would have to be predefined, and also redefinable, in any class). In that case the sender (caller) could detect and handle asynchronous communication failures. However, in languages with both forms of processor communication (as proposed in this article), this approach interferes with the shared memory communication mechanism (rule 8), because it would require that both communicating processors were somehow dedicated to asynchronous communication.

Considering all this problems, the best solution is, most likely, to provide special inter-processor direct communication functions (synchronous), which – if the caller desires – can be used to query the message communication status (processed, failed, etc.). Those special functions would have to be provided by the language library and be available in any class.

5 MP-EIFFEL APPROACH TO CONCURRENCY

MP-EIFFEL is a concurrent object-oriented programming language based on the concepts and solutions presented in the previous sections. It is characterized by the following properties:

- exclusion safety (aiming, in the future, to become safe in other synchronization aspects);
- heterogeneous processor mapping;
- readers-writer concurrent access semantics to objects;
- shared-memory synchronous processor communication;
- message passing synchronous and asynchronous processor communication;
- absence of inheritance anomalies (inheritance safety);
- disciplined concurrent exception handling;
- external concurrency control language, to specify processor mapping, scheduling policy, and other timing constraints[6].

The decision to use EIFFEL as the base language for implementing concurrency mechanisms was based on EIFFEL's simplicity, safety, coherent integration of its language mechanisms, its static type system and the support for Design by Contract. It was also decided that MP-EIFFEL should (as much as possible) contain EIFFEL, to allow the reuse of its classes.

Shared and remote objects

In MP-EIFFEL the normal object communication scheme follows the processor shared memory model, so objects might be executed by several processors. To ensure a safe usage of objects the EIFFEL's type system was augmented with two new concurrent type modifiers: SHARED and REMOTE. Object entities, depending on its concurrent type modifier, can hold references to objects with three possible concurrent scopes: normal, shared, or remote.

If an entity is neither shared or remote, then it is named normal. Normal entities can only contain expanded objects, or references to normal objects. Normal objects are created by applying the creation instruction to normal entities, and are said to belong to the creating processor.

A shared entity can only contain shared objects. An object is shared, if it was created by the creation instruction on a shared entity, or if it is a normal valued feature (attribute or function) reference of a shared object. Shared objects can be freely used by several processors. If a routine of a shared object has side-effects to the object's state ("impure"), then its use requires mutual exclusion ("writer" access mode), otherwise a "readers" access mode applies.

Remote entities can contain references to normal objects belonging to other processors. Those objects are said to be remote to the current processor. Only

[6]Not presented in this article

side-effect-free ("pure") valued features can be called on remote objects (the use of commands is forbidden). The use of remote objects follows a "readers" access mode.

The concurrency access of a type – normal, shared, or remote – will be called its concurrent scope.

"Traitors"

To ensure the safety in the MP-EIFFEL's type system, it is necessary to guarantee the absence of "traitors": objects attached to entities with different concurrent scopes.

Aiming for that goal, two type rules are imposed:

Concurrent scope propagation rule: the type of normal reference valued features and routine formal arguments maintains the concurrent scope of current object;

Normal objects passing rule: routines of non-normal objects can only receive references to normal objects if, and only if, the corresponding routine formal argument is declared as remote.

Concurrent preconditions

Preconditions in MP-EIFFEL behave as presented in the previous section. Its normal concurrent scope part is checked first (unless the causality of the logical expression, makes that impossible by the use of the logical operators OR ELSE or AND THEN); then, its non-normal part (if any, and if it involves the use of objects) is used as a conditional synchronization test. However, no synchronization is required for testing the "Voidness" of non-normal references.

This behavior requires that the non-normal part of preconditions cannot be disable by the compiling system (the normal part behaves as in EIFFEL).

Triggering: a direct inter-processor communication mechanism

Based on the reasons presented in the previous section, MP-EIFFEL also possesses a direct inter-processor communication mechanism. This mechanism is called "trigger".

A processor is able to receive trigger messages, if it owns objects with the desirable messages listed in their trigger accept clauses. The trigger accept clause is attached to classes, and has a similar structure as creation clauses, using instead a new keyword: TRIGGER.

For example a deferred class with the ability to receive external time events on behalf of its creation processor, could be defined as follows:

```
deferred class TIMER_RECEIVER
trigger
  tic
feature
  tic is
    deferred
    end;
end -- TIMER_RECEIVER
```

Triggers are inherited by descendant classes (renaming works smoothly with this mechanism since both use feature names).

Trigger messages can be send by any processor, and its structure is almost identical to a normal feature call with the difference that the call is preceded by the keyword TRIGGER (the same keyword is used in the trigger accept clause and in the trigger send instruction, because the word has both meanings). For example to trigger a tic in a processor owning a TIMER_RECEIVER object:

```
trigger a_timer_receiver.tic;
```

If the trigger message is a procedure (as happens in the above example), then the communication will be completely asynchronous, otherwise it will be synchronous (the message sender processor will wait in the point of the trigger, until the result arises).

MP-EIFFEL ensures that no trigger message will ever be lost. To implement this behavior, a receiving queue is attached to any processor that may receive trigger messages. Also, by default, all messages are ordered in "time" (using a scheme such as [Lam78]) so the normal (default) behavior of the queue is to serve oldest messages first.

Locking the access to multiple objects

To lock (with readers-writer semantics) multiple objects MP-EIFFEL uses a similar approach as SCOOP: all non-normal actual arguments are locked within the routine's body execution.

Processors lifecycle

Processors are created by applying the creation instruction to remote entities. They may have an agenda to fulfill (their creation feature), and may also have to process triggers. Their lifecycle goes from creation through a succession of "active" and "idle" states until their termination when they cease to be necessary.

The "idle" states only exist if the processor owns objects with trigger accept clauses. A processor without trigger accept clauses will terminate as soon as its creation procedure ends.

The program will end when all non-terminated processors have empty trigger queues.

Exception handling

MP-EIFFEL implements the disciplined synchronous exception handling described in the section 4 for both forms of synchronous communication: inter-object communication involving several processors and the triggering of valued features.

For the MP-EIFFEL's asynchronous form of communication – the triggering of procedures – the scheme proposed in the section 4 is implemented. The status of the asynchronous communication attempt, is verifiable by special direct inter-processor communication functions (synchronous) provided by the MP-EIFFEL's library.

Final remarks

This language adds three new keywords to EIFFEL: SHARED, REMOTE, and TRIGGER; and the class structure is augmented with a trigger accept clause (trigger export mechanism).

All mechanisms were designed taking in consideration all of the language design rules presented in the previous sections (9 rules). In particular they are quite simple to understand and use, complete, and orthogonal to each other (the example presented below is a good example of these characteristics).

Nevertheless, the precise definition of the external concurrency control language, and the compiling and run-time supporting system necessary to ensure the nonexistence of deadlocks and other liveness situations, is yet to be carefully thought and implemented.

Example: The Dining Philosophers problem

No article on concurrent programming languages is "complete" without presenting the classic dining philosophers problem proposed by Dijkstra [Dij72, Dij77]. In this problem, several philosophers (five) do their living either thinking or eating spaghetti. However, in order for a philosopher to be able to eat spaghetti he needs two forks (the ones in his left and right sides), but the number of forks is restricted to exactly the same number of philosophers (so it is necessary to share forks, and is also necessary to avoid deadlock or starvation situations). One possible solution in MP-EIFFEL is presented below.

```
class DINING_PHILOSOPHERS

creation
  make

feature

  make(num_phil,life_time: INTEGER) is
    require
      num_phil > 1;
      life_time > 0
    local
      forks: ARRAY[shared FORK];
      philosophers: ARRAY[remote PHILOSOPHER];
      a_fork: shared FORK;
      a_philosopher: remote PHILOSOPHER;
      left_fork_num,right_fork_num: INTEGER
    do
        -- creating forks...
      !!forks.make(1,num_phil);
      from n := 0 until n = num_phil loop
        n := n + 1;
        !!a_fork;
        forks.put(a_fork,n)
      end;

        -- creating philosophers (new processor for each)...
      !!philosophers.make(1,num_phil);
      from n := 0 until n = num_phil loop
        n := n + 1;
        left_fork_num := n;
        right_fork_num := (n \\ num_phil) + 1;
        !!a_philosopher.make(forks @ left_fork_num,forks @ right_fork_num);
        philosophers.put(a_philosopher,n)
      end;

        -- starting the interesting philosophers life...
      from n := 0 until n = num_phil loop
        n := n + 1;
        trigger (philosophers @ n).life(life_time) -- asynchronous message
      end;
    end;

end -- DINING_PHILOSOPHERS



-- Nothing to do here
class FORK

end -- FORK
```

```
class PHILOSOPHER

creation
  make

trigger
  life

feature

  make(left_fork,right_fork: shared FORK) is
    do
      l_fork := left_fork;
      r_fork := right_fork;
    end;

  life(life_time: INTEGER) is
    local
      t: INTEGER;
    do
      from t := 0 until t = life_time loop
        t := t + 1;
        think;
        eat(l_fork,r_fork)
      end
    end;

feature  {PHILOSOPHER}

  l_fork,r_fork: shared FORK;

  think is
    do
      print_line("I'm thinking, therefore I am...");
      ms_sleep(100 + random_integer(400)); -- sleeps [100-500] milliseconds
    end;

  eat(left_fork,right_fork: shared FORK) is
      -- Waits until both forks are available!
      -- (The implicit synchronization machine can avoid starvation and
      --  deadlocks. The former by imposing a fair resource access
      --  scheduling scheme, and the latter by taking advantage of the
      --  fact that both locks are requested in the same place)
    do
      print_line("I'm eating, therefore I might continue to be...");
    end;

end -- PHILOSOPHER
```

6 CONCLUSION

In this article a new concurrent object-oriented programming language – MP-Eiffel – was introduced, taking special care to present the reasoning behind the choices made throughout all its mechanisms and their semantics.

A prototype compiler is being developed which will allow its practical validation and experimentation. In the future, the concurrency control language will be implemented, taking into consideration the absence of deadlocks and other liveness synchronization problems.

REFERENCES

[Agh86] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[AK99] Gul A. Agha and Wooyoung Kim. Actors: A unifying model for parallel and distributed computing. *Journal of Systems Architecture*, 45(15), September 1999.

[Ame87] Pierre America. Pool-t: A parallel object-oriented language. In Akinori Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[Car88] L. Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–79. ACM Press, 1988.

[Car89] Denis Caromel. Service, Asynchrony, and Wait-by-Necessity. *Journal of Object-Oriented Programming*, 2(4):12–18, 1989.

[Car93] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[CHC90] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135. ACM Press, 1990.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[Coo89] William R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):305–311, 1989. Originally in Proc. European Conf. on Object-Oriented Programming (ECOOP).

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[Dij68] Edsger W. Dijkstra. *Cooperating Sequential Processes*. Programming Languages. Academic Press, New York, 1968.

[Dij72] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, pages 72–93. Academic Press, 1972. EWD310.

[Dij77] Edsger W. Dijkstra. Two starvation-free solutions of a general exclusion problem. circulated privately known as EWD625, 1977.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[Hab69] A. N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–377, 1969.

[Han72] Per Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, 1972.

[Han93] Per Brinch Hansen. Monitors and concurrent pascal: a personal history. In *The second ACM SIGPLAN conference on History of programming languages*, pages 1–35. ACM Press, 1993.

[Han99] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, 1999.

[Hoa73] C. A. R. Hoare. Hints on programming language design. Technical Report STAN-CS-73-403, Stanford Artificial Intelligence Laboratory, Computer Science Department, Stanford University, 1973.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[KL89] Dennis G. Kafura and Keung Hae Lee. Inheritance in actor based concurrent object-oriented languages. In *Proceedings of the Third European Conference on Object-Oriented Programming*, July 1989.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[LN78] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. In *Proceedings of the Second International Symposium on Operating Systems*, October 1978. reprinted in Operating Systems Review, Vol. 13, No. 2, April 1979, pp. 3-19.

[LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, Englewood Cliffs, N.J., March 1992. 2nd printing.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[Par72a] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par72b] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

[RA95] Shangping Ren and Gul A. Agha. Rtsynchronizer: language support for real-time specifications in distributed systems. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 50–59. ACM Press, 1995.

[RF77] Manfred Ruschitzka and R. S. Fabry. A unifying approach to scheduling. *Communications of the ACM*, 20(7):469–477, 1977.

ABOUT THE AUTHORS

Miguel Oliveira e Silva is a PhD student and teaching assistant in the Electronics and Telecommunications Department of the University of Aveiro, Portugal. He is also part of the IEETA's research group in the same University. He can be reached at mos@det.ua.pt. See also http://www.ieeta.pt/~mos.