

Concurrent object oriented programming with Asynchronous References

Pierre Vignéras
pierre@vigneras.name

April, 13th 2007

Abstract

In the very next future, applications will have to be highly concurrent to get some speedup from the new trend of chip-multithreading processor architecture. But writing concurrent applications is provably more difficult than developing sequential ones.

We define three objectives a concurrent programming model should target to be massively adopted by the developer community. We propose such a model for imperative object oriented languages based on an extension of the standard synchronous reference called *Asynchronous Reference* that naturally provides the *Asynchronous Method Invocation* paradigm. Our model, though simple, has many interesting side effect issues such as inter- and intra-object concurrency degree, asynchronous policy mapping, asynchronous policy sharing, non-determinism proportion and exception handling.

Our proposition can be seen as a base brick on top of which many different concurrent frameworks can be built upon. The open-source Mandala framework includes a Java implementation of our proposition.

Keywords: Concurrent object oriented programming, Asynchronous method invocation

1 Introduction

Computer Science is currently under a major revolution [48, 5]: processor manufacturers have changed the way they will follow the Moore's law [38]. The doubling of transistor number per chip every 24 months is no longer used to make new processors faster but to embed more processors on the same chip. This turn over has a huge consequence for software developers: sequential applications can no longer profit from the traditional doubling in performance consequence of the Moore's law — the free lunch is over [47]. The message is clear: softwares will have to be concurrent to get some performance gain from new hardware.

Concurrent programming is not new and many research results have already been published about it. One of them is the very influential Amdahl's

¹It is often mentionned that the law is about doubling the number of processors per chip every 18 months for unknown reasons.

law¹ which basically states that for a given parallel architecture, a significant speedup can be achieved only by executing programs with a sequential part kept to a strict minimum. Hence, not only programs will have to be concurrent, they will have to be *highly* concurrent.

Unfortunately, despite many research in the area of concurrent programming, the vast majority of developers are still not very confident with it. Many reasons can be given: the traditional studied computing model — the Turing Machine — is inherently sequential, and most widely adopted languages are based on it. Another reason is that concurrent programming is more difficult than sequential programming mainly because of the non-determinism it introduces. Therefore concurrent programming is rarely taught as a core course despite the recent adoption of concurrent constructions in main stream languages such as Java [19] and C# [14]. Of course, one may argue that those constructions — threads and locks — are the worst available [48, 27, 22] and that many other models [36, 37], paradigms [51, 25, 30] and languages [41, 43] specifically designed for concurrency have already been proposed. The fact remains that none of them have been massively adopted by the community of developers. As discussed in [27], developers seem to be regrettably more guided by syntax than semantic.

Therefore, we should not try to replace existing languages by others. Since the vast majority of mostly used languages are Object Oriented Languages², we propose in this paper, a new paradigm for Concurrent Object Oriented Programming (COOP). We define three dimensions on which COOP propositions can be mapped to in order to analyse their potential of massive adoption by the community of developers:

usability: legacy code can be used concurrently without any modifications;

flexibility: any model of concurrency can be implemented;

extensibility: implementations for specific cases — such as the distributed one — remains possible.

This paper is organised as follow: next section presents related works, section 3 presents our contribution, section 4 considers our model from a performance perspective and section 5 finally concludes.

2 Related Works

In the imperative, non-object oriented world, process (or thread) is the traditional paradigm used to express concurrency. The `fork()` function is the earliest proposed construct to initiate concurrency at the language level³. Synchronization is usually provided by the paired `join()` function. In the survey of Michael Philippsen [41], the expressive power of any

¹It may be argued that the Gustafson's law should be used instead but Shi has proved in [45] that they are rigorously identical.

²See <http://www.tiobe.com/tpci.htm> for details.

³The syntax may vary, especially in the case of threads where we can find `pthread.create()`, `clone()` and the like. The underlying mechanisms are nevertheless basically the same.

other paradigm specifically designed to initiate concurrency is systematically compared to the expressive power of the general `fork()/join()`. It seems that this general paradigm, even if it has its own disadvantages, at least provides a very expressive power, probably the best one.

In the object oriented world, function calls are replaced by the method invocation paradigm. Therefore, it is natural to extend it to asynchronous method invocation (AMI) to provide concurrency naturally. Fortunately, in his survey, Michael Philippsen stated that AMI is as expressive as the `fork()` statement. That paradigm is not only natural, it is also very efficient to express initiation of concurrency.

2.1 Active Objects

The active object paradigm decouples method invocation from method execution. This way, a method invocation on an active object will immediately return a result, generally called a *future* [20], which will provide the necessary means to obtain later the real result of the method execution. This execution will take place asynchronously in a thread dedicated to the corresponding object. This principle can be found in several works and has been described as a design pattern in [25].

We will insist here on the implementation of this principle in Java through the ProActive platform [7] which we call active objects *à la* ProActive, and first described by Denis Caromel in [8]⁴. These active objects have several characteristics:

1. an active object *à la* ProActive is associated with only one single thread; this thread is responsible for sequentially⁵ invoking the called methods on the actual object;
2. method calls on active objects are syntactically regular calls but still asynchronous (and possibly remote);
3. *future* results are polymorphic with the declared return type of the corresponding method and use a mechanism called *wait by necessity*: when a method of the future is invoked, the caller is blocked until the associated result is actually available, *i.e.*, until the asynchronous method call is finished.

As for us, the main problem of this paradigm is the low degree of concurrency reachable since active objects can only have a single thread. More than the performance issue, this constraint is subject to deadlock. Consider two active objects referencing each other recursively as a case study.

A criterion has been proposed [6] to decide when an object can become active without changing the application behavior: basically, the *accessibility set* of an active object should be closed, that is all the objects it uses should only be used by itself. Even if, as a consequence, this criterion solves the deadlock problem, it restricts users to a subset of the Java language

⁴The first platform was written in the Eiffel programming language and was known as Eiffel//.

⁵The default scheduling of calls is FIFO but this can be changed to suite application specific needs.

where threads in particular are unavailable. Without the help of an automatic tool, it is rather difficult to develop under such constraints since some classes may use threads without mentioning it in the API⁶. The criterion has thus been extended to a multi-threaded context [10] but the constraints are too high to be usable in practice.

The second characteristic presented above also needs some clarifications. Being able to express an AMI as a standard synchronous method invocation without any syntax modification is what we call *fully transparent* AMI. Consider the prog. 2.1.

```
Result r = myObject.myMethod(myArgs);
SomeLibrary sl = new SomeLibrary(r);
sl.doSomething(); // maybe with r
```

PROG. 2.1: Fully Transparent AMI.

Nothing clearly identify any AMI in this code. If the variable `myObject` refers to an active object, then the call `myObject.myMethod(myArgs)` becomes a fully transparent AMI. The caller thread does not have to wait for the completion of the execution of the called method, and it can immediately proceed to the next instruction.

This feature may be looked at an objective that any COOL should achieve at a first glance. However, it has a major drawback: what happens if the called method throws an exception? Exceptions and AMI is an open issue discussed in section 3.5.

The third characteristic — polymorphic futures — is obviously related to the second characteristic: fully transparent AMI. In the example code considered, the return result is a polymorphic future, a subtype of the `Result` type. The caller proceeds directly to the next instruction. If we guess that the call to the constructor `new SomeLibrary(r)` may only assign the future `r` to one of its field, nothing prevents the method call `doSomething()` to use it — *i.e.* execute an `r.m(...)` call. By implementing the wait-by-necessity mechanism, any method call on a future blocks the caller until the result becomes available. Unfortunately, this solution has also a major drawback: in the vast majority of legacy code⁷, the result of a method invocation is used immediately after the call. Hence, for legacy code, the wait-by-necessity mechanism does not lead to improved performance, it may even decrease it because of the inner concurrency mechanism overhead. This is the reason why we put ProActive in the *weakly concurrent* category of figure 1. Concurrency degree will be discussed in section 3.4.1.

Implementations and drawbacks of fully transparent AMI have already been discussed by the author in [54].

Also, in the case of a remote call, the client is blocked until the server acknowledges that it has taken the asynchronous call into account (*rendez-vous*). This constraint does not fulfill our extensibility objective: it may

⁶The Sun implementation of the `java.security.SecureRandom` is such a class.

⁷A deep analysis of the JDK is currently an ongoing work. It will give a quantitative metric of this statement.

be desirable for performance reasons to provide client-side asynchronous call [42, 11]. We will further discuss the remote extension of the AMI paradigm in section 3.7.

2.2 Actors

The actor concept was introduced by Carl Hewitt in 1977 [21]. A calculus model was defined by Gul Agha and Carl Hewitt in 1986 [3, 4]. The term *actor* can then be used to deal with the original meaning or with the precise semantic of the model. Basically, actors are active objects which exclusively use asynchronous communications. In such a system, all objects are active. Communication is done through the use of a mailbox system. The mailbox system guarantees the reception of messages in an undefined order. Actors can make limited actions:

- send a message to actors (including themselves);
- create new actors;
- specify a *replacement behavior*.

A behavior handles only one message. At any given time, an actor has a current behavior. The current behavior specifies the replacement behavior, *i.e.* the behavior which will handle the next message.

An extension to the Java language has been proposed [51] but, to our knowledge, has never been implemented. This model (and all homogeneous models of active objects) falls in the *highly concurrent* category of figure 1. It provides a clear separation between the concurrent behavior of an object and its public interface (type) [18] providing high extensibility. But the actor model does not integrate neatly with the OO paradigm and thus makes the use of legacy code difficult. One reason — among many [41] — is that an actor can become a new actor with a totally different behavior. It is therefore difficult to define a type system, especially one that allows subtyping as required in an OO paradigm.

2.3 Separates

The model of separate objects was introduced by Bertrand Meyer [30] as an extension to the Eiffel language allowing the systematic asynchronous execution of procedural invocations (methods declared returning *void*). A Java implementation of his work has been proposed [23]. We will not further discuss this model because it is constrained to procedural methods: on 17254 methods of the JDK v1.4⁸, only 34 % are procedures. Among these, many have no interest to be executed asynchronously but are still used very often:

```
Object.[wait|notify|finalize](),  
Thread.[start|interrupt](), *Stream.[close|flush], etc.
```

⁸Only the classes prefixed by `java` was considered.

For this reason, this model falls in the *weakly concurrent* category of figure 1.

Procedural invocations supports *one-way messaging* which obviously leads to the minimum of messages needed to be exchanged when we extend the AMI paradigm to the remote case. However, in their exploration of the language design space, Papathomas *et al.* [40] showed that even if *one-way messaging* passing is expressive, it is undesirable because higher-level request-reply protocols must be explicitly programmed which reduces software re-usability and is by the way potentially more error-prone.

2.4 The E language

The E [32] language is a distributed, persistent, secure programming language designed for robust composition of concurrent and potentially distributed code [35]. This work is probably the closest to us as it is also based on an extension of the standard synchronous reference. Basically, in E, any reference can be used to call a method asynchronously — an *eventual send* — in the E vocabulary. But only one type of reference — the so called *near reference* — supports the usual synchronous method invocation — an *immediate call* in the E terminology. The result of an AMI is a kind of future — called a *promise* — where waiting for the availability of the result is not possible. Event driven programming is thus enforced: the developer registers a sort of a callback that will be automatically called when the requested AMI terminates. The important side effect of this restriction is that deadlock cannot happen *by design*. Nevertheless, other forms of liveness issue still remain, as we will further discuss in section 3.4.4.

E objects are aggregated into persistent process-like units called vats. A vat is responsible for the handling of every requests made on the objects it aggregates. A standard stack is used for synchronous calls whereas a queue is used for asynchronous ones. The design guarantees that for any inter-vat reference, only AMI can be performed.

This framework is very sound but it does not fulfill some of our requirements:

1. it defines a new language and we have seen that despite many other languages designed for concurrency, they have not been widely adopted. We should notice however that E can reuse directly the vast majority of Java classes. This may definitely help in its massive adoption.
2. Strong encryption is used for inter-vat communication; this level of security is unnecessary in the case of cluster computing for example.
3. A single thread is available in one vat to deal with all the method invocations made on every aggregated objects. This leads to a concurrency degree that is far worse than in the active object paradigm case (without full transparency). It is possible to increase the concurrency degree of the E application by distributing its objects among many vats. But then the required encrypted communication may lead to undesirable overhead. This is why E fits in the middle class as far as concurrency degree is concerned on figure 1.

2.5 Active Containers

The active container model has been defined by the author in [12, 11]. It provides an abstraction where objects are accessed asynchronously through a hash map. The advantage of this concept is that, while defined in the local case, it extends naturally to the remote case, supporting many extra features such as remote instantiation or object migration. An open-source Java implementation is available in [52]. Since the active containers model is one possible implementation of our proposition, we will see that it can fit in any category.

2.6 Other works

Many other works deal with concurrency: Mozart [2], Erlang [1], Haskell [22] are few examples but they rely on very different languages that have still not been adopted for various reasons. Other works include CORBA [44, 55], ARMI [15] or RRMI [50] but they are often designed specifically for the remote case. We believe that AMI must first be well defined in the local case to be extended to the remote case. We will show how our proposition, though defined for the local case, extends naturally to the remote case.

3 Proposition: the Asynchronous Reference Model

Developers are used to the reference concept. While objects remain in memory, references are used to invoke their methods. Extending this reference concept to provide an orthogonal aspect is natural and not new. Two extensions are already provided for two very different aspects:

Remote Aspect: CORBA [44], DCOM [31], or Java/RMI [46] propose *remote references* to extend more or less transparently the local synchronous method invocation into a remote one;

Memory Aspect: Java provides four different sort of references (strong, soft, weak and phantom) to manage the relationship between a given object and the garbage collector⁹.

Our proposition is to provide a new extension of the reference concept that will cover the concurrent aspect providing asynchronous method invocation. It should be seen as a basic brick on top of which richer frameworks can be built — what we called the flexibility and extensibility objectives. Especially, it does not provide any form of transparency. Transparency and AMI is already discussed in [54].

3.1 Basics

An asynchronous reference (AR) is just an extension of the usual reference concept where method of objects can be invoked asynchronously.

⁹Even if in that specific case, it is not possible to invoke directly a method through such a reference.

3.2 Result Recovery

In the synchronous case, the way a result is recovered is clear. The caller is (virtually) blocked¹⁰ until the result becomes available. Then the caller resume its execution to the next instruction. Since in the asynchronous case, the caller continues its execution, some mechanisms should be provided to recover the result. Many solutions have been proposed in the past. In the case of AMI, we have the following choices:

Polling: the availability of the result is polled in some ways (should have the possibility to do something else in between);

Waiting: the caller¹¹ is blocked until the result becomes available (should have the possibility to do something else before the waiting);

Callback: a special object — a callback will be notified automatically when the result becomes available.

Traditionally, the future concept [20] is adapted to provide polling and waiting. In our case, we will define a future as a composition of different *facets*:

InvocationInfo facet: provides information on the AMI such as the method name, the list of arguments or the target object;

MethodResult facet: provides the result;

InvocationObserver facet: provides the polling facility;

InvocationEventsWaiter facet: provides the waiting facility;

Cancelable facet: provides a way to cancel or interrupt an AMI.

It is very important to decompose a future from these very specific facets as we can compose other objects when reducing the number of facets. For example, a promise, as defined in the E language is a kind of future where waiting is forbidden. In our system, a promise is just the combination of all preceding facets but the `InvocationEventsWaiter` one. When a *callback* is used for the result recovery, it is unnecessary to provide `InvocationObserver`, `InvocationEventsWaiter` and `Cancelable` facets.

Facets and their combinations should be first class objects to be transmitted to other objects. After an AMI, a client may ask another object to deal with the result, sending him the future, the promise or any other combination of facets.

¹⁰Obviously, in the local case, the execution thread is not blocked at all. But we may consider this as an implementation detail. Note that in the remote case, the caller is definitely blocked, and still remote method invocation remains synchronous.

¹¹Actually it is not required that the caller is the one who wait, it can delegate as we will see below.

3.3 Asynchronous Method Invocation

When a client wants to perform a method invocation, he should provide what we call a *method call specification* containing all the required informations to perform the call. In the synchronous case, the method and the arguments of that method are enough. In the asynchronous case, we add two other informations:

- the callback that should be used when the method execution ends (we do not differentiate a normal return and a thrown exception at that stage);
- a meta object that holds informations relevant to the *asynchronous policy*¹² used by the asynchronous reference.

This last point is quite important and needs some further clarifications. At the upper layer, where an asynchronous method invocation is *expressed* through a more or less transparent mechanism, the programmer can consider two different levels:

AMI Syntax Level: Whatever the actual concurrency implementation layer is, the method specified will eventually be executed later on, and the callback, if specified, will be called afterward;

AMI Semantic Level: Other meta characteristics of an asynchronous method invocation.

The syntax level provides a clear understanding of what is the basic feature provided: when the future result is returned, the caller can continue its execution; its request will eventually be processed at one time or another. As we will discuss in 3.4.1 the syntax level plays a significant role in the overall concurrency degree of an AMI-based program¹³.

At the semantic level we find characteristics that have an influence on the way the implementation layer will execute the request including:

Priority: a priority is given to an AMI;

Recorded Type: specify the way the call should be recorded in the list of pending calls (for example stacked, queued, ...);

Parameter Passing Policy: it can be of a great importance to know or to specify whether parameters should be passed by copy or by reference;

Condition: a condition should be evaluated to `true` before the execution of the specified method — this leads to guarded method style [29, 16];

Reply: the implementation may provide a one-way facility for performance reason, in this case, no result will be returned.

¹²In the lack of any ambiguity, the term “policy” will implicitly means “asynchronous policy” in the rest of this document.

¹³In this document, “AMI-based program” refers to a program written using the asynchronous method invocation paradigm.

The meta object that is provided in the method call specification is relevant only to the policy of the underlying implementation. It is up to that policy to interpret any information provided in a meta object. For example, a policy may ignore completely a meta parameter. So when a meta is specified in an AMI, the caller may want to know the actual asynchronous policy that is attached to the asynchronous reference.

This leads us to the simplistic interface given in prog. 3.1.

```
public interface AsynchronousReference<T> {
    <R> FutureClient<R> call(MethodCallSpec m);
    AsynchronousPolicy getAsynchronousPolicy();
}
```

PROG. 3.1: The first-class object asynchronous reference interface definition.

3.4 Asynchronous Policy

We have seen in the preceding section that when a client is performing an AMI through `AsynchronousReference.call()`, it provides a meta object in the method call specification that is only relevant to a specific policy. For example, specifying a priority may be irrelevant if methods are selected for their execution with a sequential but random policy.

Hence, we suggest that this meta object should be created directly by the attached policy. This leads us to the interface `AsynchronousPolicy` defined in prog. 3.2.

```
public interface AsynchronousPolicy {
    Meta newMeta(Object... args);
}
```

PROG. 3.2: The asynchronous policy interface definition.

3.4.1 Concurrency Degree

It should be clear that when an AMI is requested, it does not necessarily mean that a *callee* thread will run the specified method concurrently with the *caller* thread. The *caller* may terminate its execution while the *callee* may not have even begun the execution of the specified method. However, AMI helps expressing concurrency and generally produces concurrency up to a certain degree. Determining that degree precisely for any AMI based program is beyond the scope of this document and is one of our future works.

Intuitively, the concurrency degree of a program P is a function of:

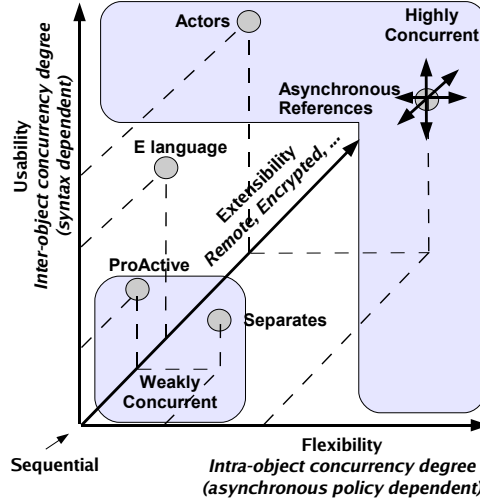


Figure 1: The three dimensions in COOP.

the *inter-object concurrency degree* represents the amount of concurrency between different objects in P . It only depends on the syntax of the program: “*how often*” AMIs are requested in P basically. If waiting for the result is allowed then we should also evaluate “*how far*” (measured in unit of instructions), from each AMI, the result is waited for.

the *intra-object concurrency degree* represents the amount of concurrency inside objects in P . This process-based concept is related to the object-based *internal concurrency* concept used in [39]. It does not depend on the syntax but rather on the asynchronous policy attached to each asynchronous reference used in P .

Figure 1 illustrates the effect of each concurrency degree on the final concurrency degree of a given AMI-based application.

3.4.2 Asynchronous Semantics

Asynchronous policies come into two different major flavors — called *asynchronous semantics*¹⁴ — that clearly influence the intra-object concurrency degree:

¹⁴In the lack of any ambiguity, the term “semantic” will implicitly means “asynchronous semantic” in the rest of this document.

Concurrent Semantic: Methods invoked asynchronously on a given object¹⁵ may possibly be executed concurrently allowing internal concurrency. Such a semantic can be implemented by a *thread-per-call policy* where a thread is created for each AMI. A policy that is using a thread pool to execute requested AMIs also falls into this category. This semantic clearly increases the intra-object concurrency degree of an AMI-based program.

Non-Concurrent Semantic: Methods invoked asynchronously on a given object are never executed concurrently preventing internal concurrency. Such a semantic can be implemented by a single thread that is taking requests from a queue — as in the active object paradigm. This semantic clearly decreases the intra-object concurrency degree of an AMI-based program.

In the actor model, the active object model and the E model, we have seen that a non-concurrent semantic is used. This is the reason why they are close to the y-axis on the figure 1.

3.4.3 Correctness

One may believe that a given asynchronous reference can be used to refer to an object and later on, to another one as it is naturally done in the synchronous case using the assignment operator. Unfortunately, this is not true. Of course, the type of the declared reference should be a super-type of the object type it is referring to.

But there is another condition. Since some objects are thread safe, they can be referred to by asynchronous references linked to any policy that implements the concurrent semantic. Of course, this is not true for non-thread-safe objects.

To find some solutions to this (re-)assignment problem, we first have to define a new concept:

A correct asynchronous policy for a given object is one that guarantees both safety and liveness in the context of concurrent requests through an asynchronous reference.

Safety and liveness are two opposite design forces in COOP defined in [26] as:

Safety: nothing bad ever happens to an object;

Liveness: something eventually happens within a thread.

Therefore, the other condition for an asynchronous reference to be assigned an object to refer to is that the asynchronous reference should be linked with a correct policy for the target object.

Note that given a correct policy for a given instance of a class C, nothing can be said about the *correctness* of the same policy for any instance of any

¹⁵It is important to specify methods on a given object since policies can be *shared* as we will see in 3.4.5.

subclass of C in general. Note also that given a correct policy P for an object \circ , it cannot be said that any subclasses of P are also correct for \circ in general. This is directly linked to the unsolvable inheritance anomaly issue [13].

3.4.4 Safety *versus* Liveness

When any object can be used asynchronously, both type of semantic seem to be required: some legacy objects are not *thread safe*, therefore, they can only be used asynchronously with a policy that implements the non-concurrent semantic to ensure safety. On the other end, if only non-concurrent semantic were available, not only would it limit the concurrency degree of AMI-based program but it would lead to the deadlock issue already mentioned in section 2.1.

As always in COOP, a good tradeoff should be found between safety and liveness. The term “good” refers to the *quality* of the program which is itself a compromise between *re-usability* and *performance*. Many works have been done in the past to find such a good tradeoff. For example, when safety is guaranteed using exclusively a non-concurrent semantic, at least two different options ensure liveness:

- enforcing the use of a constraining criterion such as the one briefly described section 2.1 about the active object paradigm, it can be proved that an AMIs-based program is deadlock-free;
- preventing the use of any blocking method and relying only on callbacks (event-driven programming) as in the E language presented section 2.4.

On the other side, when liveness is to be guaranteed (using possibly both concurrent and non-concurrent semantics), safety can be ensured using some form of transactions where the set of modifications made on an object by an AMI is both indivisible and recoverable. If this idea is not new (see Argus [28] for example), it is definitely the new trend [22]. Note that in this case, deadlocks can still be encountered but it is no more an issue since some external events (timeout, deadlock checker, or even the final user) will abort the transaction reverting the status of the object to a consistent state.

However, deadlock is not the only liveness issue in an AMI-based program. Livelock, gridlock and datalock are defined on the E language web site [33]. If livelocks are theoretically unavoidable (halting problem), datalocks are both deterministic and rare (see also [34] for details).

Nevertheless, any policy is subject to *gridlock* — also called *Resource Exhaustion* in [26]. This is due to the number of concurrent entities (thread, process) usable for the execution of asynchronous method invocations that always remains bounded either by the system¹⁶ or by the policy itself. A gridlock occurs, for a given bound n , when n waiting methods have already been invoked and taken into account by the policy. The $(n+1)^{\text{th}}$ call responsible for the notification (`notify()`) of the n other waiting calls cannot be executed.

¹⁶We also consider a limit imposed by the hardware (amount of RAM) as a system limit.

This bound is defined by each policy; it directly defines the maximum amount of internal concurrency an asynchronous reference can produce¹⁷. Its inverse $1/n$ can be interpreted as the *sensibility* of a policy to gridlock. Asynchronous policies can thus be ordered by their gridlock sensibility. For example, if:

FIFO represents a policy that is a first-in-first-out implementation of the non-concurrent semantic ($n = 1$);

Random represents a policy that is a random implementation of the non-concurrent semantic ($n = 1$);

ThreadPooled represents a policy that is a thread-pool implementation of the concurrent semantic ($1 < n \leq max$);

Threaded represents a policy that is a thread-per-call implementation of the concurrent semantic ($n = max$);

then comparing their gridlock sensibility gives:

Threaded \leq ThreadPooled \leq Random = FIFO

3.4.5 Asynchronous Policy Sharing

Unless the framework forbids *multiple* asynchronous references on the same object (using a singleton design pattern [17] for example), four basic different cases should be considered as illustrated in figure 2:

- If all asynchronous references are referring to the same object (case 2(a) and 2(b)), then:
 - sharing a correct policy (case 2(a)) is correct since the policy is the central manager of concurrent requests;
 - using different correct policies (case 2(b)) is incorrect in general as a decentralized management of concurrent requests usually requires some coordinations to be correct;
- If asynchronous references are referring to different objects (case 2(c) and 2(d)), then:
 - sharing a correct policy (case 2(c)) is incorrect in general as a policy may not be designed to deal with different target objects;
 - using two different correct policies (case 2(d)) is correct — this is the normal case.

Among the four possibilities, two are correct, and two are incorrect in general. Note that case 2(b) does conform to the intuition that an object should not be reachable through different unsynchronised paths to ensure liveness unless this object has special properties such as immutability

¹⁷Even if asynchronous policies can be shared as we will see in the next section, this bound remains the upper bound for a given object.

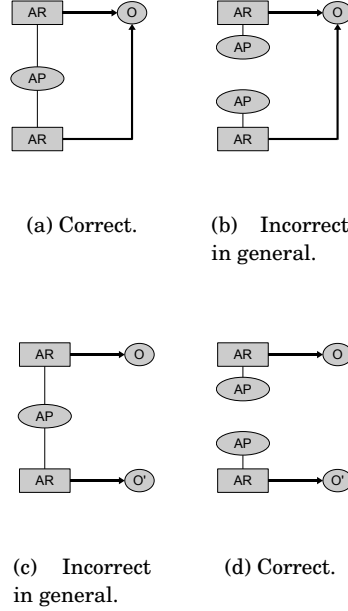


Figure 2: Sharing of Asynchronous Policy.

AR: Asynchronous Reference, AP: Asynchronous policy, O, O': objects.

or thread-safety. Since we consider standard references as special references linked to a synchronous policy, it is not correct to have both an asynchronous and a synchronous reference on the same object.

In frameworks based on the Actors computation model, each actor has its own asynchronous policy — the so called *behavior*¹⁸. So, the model applies a merge of two cases where for any reference on the same object, the asynchronous policy is shared (case 2(a)) but two distinct objects are however mapped to different asynchronous policies (case 2(d)). The Active Object case is similar but the asynchronous policy of each object is a different instance of the *same* class¹⁹ (usually a FIFO policy). Finally, in the E computation model, a policy is shared by all objects of a same set — the *vat*. Hence, a merge of two cases occurs: any asynchronous reference on any object of the same vat is using a shared asynchronous policy (case 2(a)) — a FIFO policy — and for two different asynchronous references on objects contained in different vats, two different asynchronous policies — instances of the same class — are used (case 2(d)).

The choice of one case or another (or of any combination) is usually led by the semantic guarantees the framework wants to achieve (deadlock free, in-order method calls, and so on). Note that with sharing, the probability

¹⁸This is not really accurate as in the Actors model, the paradigm is not AMI but *message passing*.

¹⁹Even if, in some frameworks (e.g. ProActive), the user can customize the asynchronous policy, it is usually restricted to composition to keep the non-concurrent asynchronous semantic guarantee.

of gridlock increases as the number of concurrent entities may be divided by the number of objects referred to by all asynchronous references linked to the same asynchronous policy. This is the reason why in the general case (with legacy objects), sharing non-concurrent semantic implementations should be avoided.

3.4.6 Non-determinism

Non-determinism is one of the principal characteristics of concurrent programs. For an AR-based program²⁰, the sharing of asynchronous policies has a direct influence on its overall non-determinism.

When sharing is not allowed, each asynchronous reference has its own policy (e.g. Actors or Active Objects). Without any coordination, each policy requests execution of method invocations to the underlying layer — usually a thread library. Hence, the global interleaving of method invocations is left to that layer. In other words, the overall non-determinism of the program does not depend entirely on it, but rather, one part of that non-determinism depends on the immediate underlying layer — which may itself rely on another layer²¹.

On the other extreme, when only one asynchronous policy is managing all method invocation requests in the application — as for E applications for example²² — a significant part of the overall non-determinism is no more left to the underlying layer but to the policy itself. If a policy is smart enough, it can schedule different method invocations to guarantee some deterministic properties.

Between these two extremes, the more policies are shared in an AR-based program, the less non-determinism is left to the underlying layer. Of course, a policy can select methods on a random basis but in this case, the non-determinism of that part of the program is the responsibility of the policy itself, not of the underlying layer.

Giving a quantitative metric for the part of non-determinism that is left on the program only is an interesting open issue to our knowledge and beyond the scope of this paper.

3.4.7 Mapping

The developer of a class *C* needs a mechanism to specify the set of all the correct asynchronous policies that can be linked with any asynchronous references used to access any instance of *C*. Such a specification can be done by a typing system and checked at compile time. Unfortunately, not only the vast majority of legacy classes do not have such a specification but it may not be possible for a developer to know in advance all the possible correct asynchronous policies that can be used.

²⁰In this document, “AR-based program” refers to a program written using the asynchronous reference model. An AR-based program is an AMI-based one but the converse is not true in general.

²¹For example when the underlying layer is a user thread library with an m-to-n mapping to kernel threads.

²²To be really accurate, this is true for E applications that are only using one *vat*.

We thus propose to postpone the mapping to a customizable factory object [17] providing a great flexibility in the decision making. For example a factory may use:

- a separate file (flat/property, database, ...) that contains a list of pairs (C, S) , where C represents the class of objects that should be accessed through any asynchronous policy in the set S ;
- an annotation given in the source code to specify the set S of asynchronous policies with which an instance of a class C can be used through;
- a combination of both previous mechanism.

It is then up to the upper framework to guarantee (or not) the correctness property when a mapping is requested.

3.5 Exception Handling

Dealing with exceptions in an AMI-based program is a known issue [24, 9]. Traditionally, exceptions are separated in two different classes:

Functional Exception: the exception has been thrown “normally” by the logic of the called method as it may have been the case in the sequential case;

Non-Functional Exception: the exception is not directly related to the logic of the called method.

Of course, since the caller of the AMI may have already died when the exception is raised by the callee, using a `try/catch` block enclosing the asynchronous method call is irrelevant (unless the caller is automatically blocked before living the catch statement). This is one of the drawbacks of fully transparent AMI as mentioned in section 3 and shown in [54].

Therefore, a simple and highly flexible mechanism is proposed: when an exception is raised, the corresponding AMI is considered terminated. It is then up to the related asynchronous policy to handle that exception according to several parameters:

Type: the handling of a functional exception may be different from a non-functional one. For example, for a functional exception the caller thread (if alive) may be notified using interruption and any attempt of result recovery will result in the exception to be forwarded up. For a non-functional exception, the policy can notify a global handler that will retry the AMI later on.

Meta: the meta object provided in the method call specification can contain informations relevant to the handling of exception. For example, according to a `critical` flag the policy may decide to block any further AMI on the same target object until the exception is taken into account.

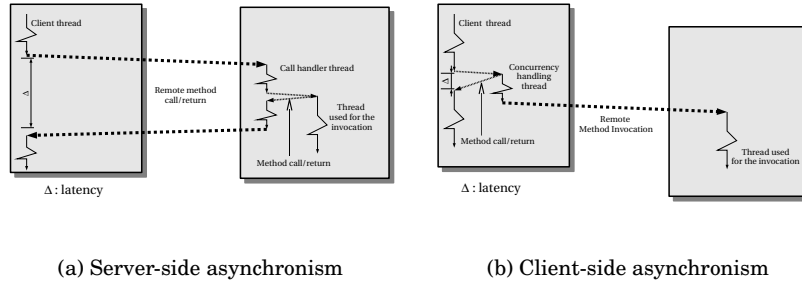


Figure 3: The two sorts of asynchronism in the remote case.

3.6 Grouped Operations

It may be tempting to provide grouped operations on asynchronous references such as *waiting the termination of all invocations*, or *cancelling every calls*. If the meaning of this facilities is clear, what are their real benefits for a *caller thread*? Such a thread may want to cancel all of *its* calls, not *all* the calls made on a given object. This distinction is very important, since an object may have many different clients especially in the remote case.

So, grouped operations should not be available through asynchronous references but a mechanism making easy grouped operations on a set of *known* calls can easily be provided using callbacks.

3.7 Remote AMI

Seeing the wide use of the remote method invocation paradigm, we guess the asynchronous version will extend easily to the remote case. Especially, AMI is closer to the underlying transport layer which uses asynchronous message passing (UDP for example) than synchronous method calls.

When considering *remote* AMI, two cases must be distinguished [42] according to the *side* where the concurrency is provided as shown by the figure 3:

Server-Side Asynchronism the concurrency is handled on the server side introducing a latency for the client which is blocked until the call has been taken into account;

Client-Side Asynchronism the concurrency is handled on the client side, where the client is not blocked, nor during the method invocation, nor during the call transmission.

When both mechanisms are used simultaneously, the asynchronism is said to be *full*.

Tanenbaum and Marteen [49] present these two cases as part of six others message oriented communication paradigms. Using their terminology, these two cases are called respectively *receipt-based transient synchronous communication* and *transient asynchronous communication*. Note that the first one — the server side — is defined as *synchronous*, even if the client

does not have to wait until its request has been processed to continue its execution.

One may want to favor client-side asynchronism for performance reasons: the network latency induced by the server-side asynchronism may limit the benefit of using the remote AMI paradigm. Nevertheless, server-side asynchronism is necessary when the guarantee that a call has actually been taken into account by the server is required. We have already seen such a constraint in the ProActive framework presented in section 2.1. Therefore, both sides of asynchronism are useful. By chaining asynchronous references, a high degree of flexibility that provides all forms of remote AMI can be reached.

3.8 Chained References

Since asynchronous references are first class objects, they can be chained. We will show that this feature is a significant improvement for extending the AMI paradigm to the remote case. But we first have to define few notations to make the discussion easier. If i is an instance of a class C , then:

- $R(i:C)$ represents a reference (either synchronous or asynchronous) on i ,
- $SR(i:C)$ a synchronous reference on i ,
- and $AR_{p:S}(i:C)$ an asynchronous reference on i mapped to the policy p of type S .

We also note $v:T=R(i:C)$ a variable v of type T referencing the instance i of class C where T is a supertype of C .

Consider now the following Java code example:

```
A a = new A();
B b = a;
```

The variable a is of type A and references the (anonymous) instance of the class A . If this instance is called i , then we note: $a:A = R(i:A)$. Naturally, we also have: $b:B = R(i:A)$ and it is clear that B should be a supertype of A for this expression to be valid. We note $x = R(i)$ when the declared type of the variable x and when the class of the instance i is not relevant. Similarly, we usually note $x = AR_S(i)$ when only the type of the policy instance mapped to the asynchronous reference is relevant.

Definition 3.1 (Asynchronous Reference Pair)

Consider a reference $r_1 = AR_{p_1:S_1}(i)$ an asynchronous reference mapped to an asynchronous policy p_1 of type S_1 and $r_2 = AR_{p_2:S_2}(r_1) = AR_{p_2:S_2}(AR_{p_1:S_1}(i))$, an asynchronous reference mapped to an asynchronous policy p_2 of type S_2 on the asynchronous reference r_1 . An asynchronous reference pair p on i composed of r_2 and r_1 is an asynchronous reference on i mapped to an asynchronous policy $p(p_2, p_1)$ of type $S(S_2, S_1)$ such that any method invocation made through p on i is wrapped to be conveyed through references r_2 and then r_1 .

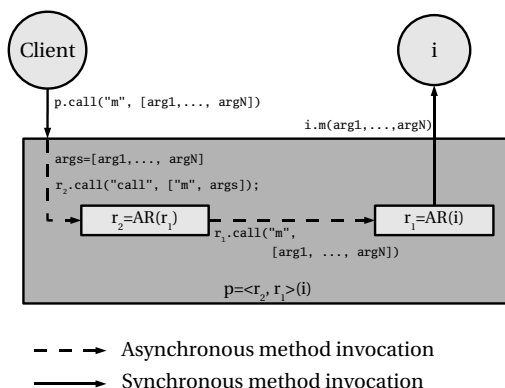


Figure 4: Calls induced by an AMI on a reference pair.

We note:

$$p = \langle r_2, r_1 \rangle (i) \equiv \text{AR}_{p(p_2, p_1):S(S_2, S_1)}(i)$$

When only asynchronous policies mapped to the references used are considered, we note:

$$p = \langle \text{AR}_{S_2}, \text{AR}_{S_1} \rangle (i)$$

Hence, an asynchronous reference pair is an asynchronous reference mapped to a special asynchronous policy that depends on both policies p_1 and p_2 . We will further discuss characteristics of this special policy in section 3.8.1.

Given an asynchronous method invocation on an asynchronous reference pair, two asynchronous calls are actually performed as shown on figure 4.

As an example of a chain of references, consider a non thread-safe object i we want to access to remotely. Suppose we have a remote implementation of the asynchronous reference concept²³. We note $\text{RAR}(j)$ a Remote Asynchronous Reference on j which provides server-side remote asynchronous method invocation (cf. section 3.7) mapped to an undefined concurrent policy x ²⁴. Using $\text{RAR}(i)$ is not a safe solution, since i is not thread-safe: the concurrent policy used by RAR may implement a concurrent semantic. So j should be a local asynchronous reference on i linked to a non-concurrent asynchronous semantic implementation such as a *First In First Out* asynchronous policy, noted $\text{AR}_{\text{FIFO}}(i)$. We then construct an asynchronous reference pair: $\langle \text{RAR}_x, \text{AR}_{\text{FIFO}} \rangle (i)$. This ensures the safe access of our object i remotely and asynchronously.

If server-side asynchronism is a problem for performance reasons, then the previous pair can also be paired with a client-local asynchronous reference to obtain *full asynchronism*. The policy — p — used by this last reference has no influence on the *safety* of the final remote object since the

²³Such a remote implementation is available in our Mandala framework [53].

²⁴This is not a so strange situation since, RMI for example does not specify how multiple connections are handled. Usually, an unbounded thread pooled is used for this purpose, but the client can not rely on it.

closest asynchronous reference to the remote object in the chain is already mapped to a FIFO policy, a non-concurrent semantic implementation. On the other hand, the policy p has a significant importance on the client side: it is responsible for the *performance* of its calls. Hence, a policy based on a *thread pool* may be a good choice (noted $p = \text{TPooled}$ below). We will further discuss the effect of chained policies in section 3.8.1. Finally, a last asynchronous reference pair makes the whole construction appearing as a single asynchronous reference. Therefore, we have:

$$\begin{array}{lll}
r1 & = & \text{AR}_{\text{FIFO}}(i) & \textit{FIFO Policy} \\
r2 & = & \text{RAR}_x(r1) = \text{RAR}_x(\text{AR}_{\text{FIFO}}(i)) & \textit{Server-side} \\
s & = & \langle r2, r1 \rangle (i) & \textit{Pair} \\
& = & \langle \text{RAR}_x, \text{AR}_{\text{FIFO}} \rangle (i) \equiv \text{AR}_{p(x, \text{FIFO})}(i) & \\
r3 & = & \text{AR}_{\text{TPooled}}(s) & \textit{Client-side} \\
& = & \text{AR}_{\text{TPooled}}(\langle \text{RAR}_x, \text{AR}_{\text{FIFO}} \rangle (i)) & \\
t & = & \langle r3, s \rangle & \textit{Pair} \\
& = & \langle \text{AR}_{\text{TPooled}}, \langle \text{RAR}_x, \text{AR}_{\text{FIFO}} \rangle \rangle (i) & \\
t & \equiv & \text{AR}(i). &
\end{array}$$

To ease the notation, the expression

$$\langle \text{AR}_{T_1}, \langle \text{AR}_{T_2}, \dots, \langle \text{AR}_{T_{n-1}}, \text{AR}_{T_n} \rangle \dots \rangle (i)$$

is simplified in

$$\langle \text{AR}_{T_1}, \text{AR}_{T_2}, \dots, \text{AR}_{T_n} \rangle (i)$$

Hence, in our example, we note:

$$t = \langle \text{AR}_{\text{TPooled}}, \text{RAR}_x, \text{AR}_{\text{FIFO}} \rangle (i)$$

Hence, chained references is a valuable solution to the issue presented in section 3.7. Both *client-side*, *server-side* and *full asynchronism* is achievable using the asynchronous reference concept.

3.8.1 Chained Semantics

We mentioned in the last section the different effect of asynchronous policies when they are linked to chained references. Resuming the last example, where we have:

$$t = \langle \text{AR}_{\text{TPooled}}, \text{RAR}_x, \text{AR}_{\text{FIFO}} \rangle (i)$$

the non-concurrent semantic is necessary from the remote object point of view, since it is not *thread-safe*. On the other end, the concurrent semantic allows clients (*caller threads*) to use a *full asynchronism* (cf. section 3.7). Hence, the chaining order has an importance that should be considered. Swapping asynchronous policies of the example, we obtain:

$$t = \langle \text{AR}_{\text{FIFO}}, \text{RAR}, \text{AR}_{\text{TPooled}} \rangle (i)$$

and the FIFO policy will have no effect on the semantic of clients asynchronous calls: they will always be concurrent²⁵. Hence, two uses of asynchronous policies can be considered:

²⁵Depending on the number of threads in the pool and the size of the pending call list.

Server Policy it ensures the asynchronous semantic used for the invocation of methods whatever policy client uses;

Client Policy it influences the efficiency of calls made by clients (latency) and the order in which they are received by the server, but not the asynchronous semantic of their invocations (concurrent or not).

Therefore, the creator of an asynchronous reference on an object is responsible for the asynchronous policy mapped. Clients can add any policy in the chain, the effect will be limited to the performance of their calls.

Chaining is an interesting concept and its use combined with sharing is an ongoing work.

4 Performance Considerations

As any sequential applications, the performance of an AR-based program depends on many factors including overall design, time and space complexity of algorithms and data structures used for example.

However, some other specific parameters should be taken into account in the case of AR-based softwares. The concurrency degree has been already presented in 3.4.1 as one of them. Sharing is another important one.

In an OO process, each live object is associated with at least one reference (synchronous or asynchronous)²⁶. Sharing asynchronous policies of some asynchronous references can thus reduce the overall memory footprint of an AR-based application and thus can improve its global performance.

Therefore, three objectives can be defined to achieve good performance of an AR-based application:

1. increasing the inter-objects concurrency degree;
2. increasing the intra-object concurrency degree;
3. increasing the number of shared policies.

Objectives 1 and 2 assume that increasing the global concurrency degree leads to better performance. This may have been argued few years ago (concurrency overhead due to scheduling and synchronization). But referring to the new trend in computer architecture towards chip multi-threading (CMT) — as discussed in the introduction of this paper — applications will have to be highly concurrent to gain from the Moore's law.

The first objective depends upon the *syntax* of the AR-based program as discussed in section 3.4.1. So either the programmer or a processing tool will have the responsibility to maximize the inter-object concurrency degree. This observation already led us to the conclusion that AMI should not be syntactically fully transparent [54].

The other two objectives do not directly depend on the syntax of the AR-based program but rather on the *mapping* of asynchronous policies to

²⁶ We do not consider objects that are not reachable anymore since either they should have been removed manually (bug) or they are candidate for garbage collection.

asynchronous references. Objective 2 depends upon the type of policies used in the AR-based program whereas objective 3 depends upon the amount of policies that are shared in the program.

Unfortunately, if aim 3 decreases the memory footprint it may also decrease the concurrency degree in some cases. Consider the extreme situation when only one non-concurrent asynchronous semantic implementation is shared in the whole program — as in the E language.

A careful static analysis may be used to help the decision of mapping and sharing a policy for a given class. For example, stateless classes can be used safely with any general policy that implements the concurrent semantic. By the way, the mapped policy can also be shared. The same conclusion can be given for any general thread safe class (classes that do not rely on any specific asynchronous policy to guarantee safety). A carefully study of this mapping problem is an ongoing work.

5 Conclusion & Future Work

We are clearly in the era of concurrent programming. Despite the proficient research in this area during the last 40 years, few results have been massively adopted by the community of developers. One reason has been identified as the root in sequential and imperative languages backed by the Turing Machine model. Thread is the *de-facto* standard for initiating concurrency. Unfortunately it does not fit well in the object oriented world. Asynchronous method invocations is a good alternative.

Our proposition — Asynchronous References — stands in this area. It is a simple extension of the synchronous reference model already well understood. It can lay the foundations of many other concurrent paradigms.

We provide an implementation in Java called Mandala [52] that supports local and remote AMI using FIFO, Random, Threaded and Thread-Pooled asynchronous policy in a fully and semi-transparent way [54].

Our proposition achieves the objectives given in the introduction:

usability: any object can be used asynchronously by being referred to by an asynchronous reference;

flexibility: asynchronous references can be linked to any kind of asynchronous policy including concurrent and non-concurrent semantic implementations;

extensibility: asynchronous references are first class objects defined by an interface that can be implemented in any specific way including the remote case.

On figure 1, *usability* is what makes our proposition plot moving up and down: the developer or a transformation tool can increase the inter-object concurrency degree by performing AMI wherever it is possible (and waiting for the result as late as is possible). *Flexibility* is what makes our proposition plot moving left and right: the developer or a tool can increase the intra-object concurrency degree by mapping asynchronous references to concurrent semantic wherever it is possible. Finally we have shown —

through the remote case — how extensible our solution is thanks to chaining. Encryption of AMI and real-time AMI are other possible extensions.

Future works include:

- finding concrete metrics of the concurrency degree of AR-based applications depending on the syntax and on the mapping of asynchronous policies (see section 3.4.1);
- finding concrete metrics of the amount of non-determinism left only to an AR-based application depending on sharing (see section 3.4.6);
- Exception handling in AMI-based programs and more specifically in AR-based ones (see section 3.5);
- Effects of chaining combined with sharing on the concurrency degree and on the overall behavior of AMI-based programs (see section 3.8);
- Automatic mapping of asynchronous policies depending on the characteristics of the target object (see section 4).

I would like to thank Pascal Grange and Omar Usman for their valuable comments and corrections.

References

- [1] The erlang programming language.
<http://www.erlang.org/>.
- [2] The moztart programming system. Web Page.
<http://www.mozart-oz.org/>.
- [3] Gul Agha. *Actors: A Model Of Concurrent Computation In Distributed Systems*. PhD thesis, University of Michigan, 1986.
- [4] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 398–407. Kaufmann, San Mateo, CA, 1988.
- [5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kart Keutzer, David A. Patterson, William Lester Plishkera, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.
[texttthttp://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html](http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html).
- [6] Isabelle Attali, Denis Caromel, and Romain Guider. A step toward automatic distribution of Java programs. In *FMOODS*, pages pp. 141–161, Stanford University, 2000. Kluwer Academic Publishers.

- [7] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. In Geoffrey C. Fox, editor, *Concurrency: practice and experience*, volume 10, pages 1043–1061. Wiley and Sons, Ltd., September–November 1998.
- [8] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.
- [9] Denis Caromel and Guillaume Chazarain. Robust exception handling in an asynchronous environment. In A. Romanovsky, C. Dony, JL. Knudsen, and A. Tripathi, editors, *Developing Systems that Handle Exceptions. Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems. Technical Report No 05-050*. Department of Computer Science. LIRMM. Montpellier-II University. 2005. July. France, 2005.
- [10] Serge Chaumette and Pascal Grange. Parallelizing multithreaded java programs: a criterion and its pi-calculus foundation. In *Workshop on Formal Methods for Parallel Programming/IPDPS*, 2002.
- [11] Serge Chaumette and Pierre Vignéras. A framework for seamlessly making object oriented applications distributed. In Gerhard R. Joubert, Wolfgang E. Nagel, F. J. Peters, and W. V. Walter, editors, *PARCO*, volume 13 of *Advances in Parallel Computing*, pages 305–312. Elsevier, 2003.
- [12] Serge Chaumette and Pierre Vignéras. Behavior model of mobile agent systems. In Hamid R. Arabnia and Rose Joshua, editors, *FCS'05 - The 2005 International Conference on Foundations of Computer Science*, Las Vegas, USA, jun, 27–30 2005. CSREA Press. ISBN: 1-932415-71-8.
- [13] Lobel Crnogorac, Anand S. Rao, and Kotagiri Ramamohanarao. Inheritance anomaly — A formal treatment. Technical Report 96/42, 1997.
- [14] ECMA International. *Standard ECMA-334, C# Language Specification*. Ecma International, third edition edition, June 2005.
- [15] Katrina E. Kerry Falkner, Paul D. Coddington, and Michael J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. Technical report, Distributed High Performance Computing Group, July 1999.
- [16] Szabolcs Ferenczi. Guarded methods vs. inheritance anomaly: Inheritance anomaly solved by nested guarded method calls. *SIGPLAN Notices*, 30(2):49–58, 1995.
- [17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN : 0-201-63361-2.
- [18] Emilio García-Roselló, José Ayude, J. Baltasar García Pérez-Schofield, and Manuel Pérez-Cota. Design principles for highly reusable concurrent object-oriented systems. *Journal of Object Technology*, vol. 1(no. 1):pages 107–123, May–June 2002.

- [19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, third edition edition, June 2005. Language: English, ISBN: 0321246780.
- [20] Baker Henry G., Jr. and Carl Hewitt. The incremental garbage collection of processes. Technical Report AIM-454, 1977.
- [21] Carl Hewitt. Viewing control structures as patterns of passing messages. In *Journal of Artificial Intelligence*, volume 8, pages p. 323–364, June 1977.
- [22] Simon Peyton Jones. Beautiful concurrency, jan 2007.
- [23] Miguel Katrib, Iskander Sierra, Mario del Valle, and Thaziel Fuentes. Java distributed separate objects. *Journal of Object Technology*, vol. 1(No. 2):pages 119–142, JulyAugust 2002.
- [24] Aaron W. Keen and Ronald A. Olsson. Exception handling during asynchronous method invocation. In *Euro-Par*, LNCS. Springer Verlag, 2002.
- [25] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc. Pattern Languages of Programs*,, 1995.
- [26] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [27] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, jan 2006. The published version of this paper is in *IEEE Computer*, 39(5):33-42, May 2006.
- [28] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst.*, 5(3):381–404, 1983.
- [29] 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.
- [30] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):56–80, 1993.
- [31] Microsoft. DCOM Technical Overview. Technical report, Microsoft Corporation, 1996.
<http://www.microsoft.com/>.
- [32] Mark S. Miller. The e language.
<http://www.erights.org/>.

- [33] Mark S. Miller. Event loop concurrency. Web Page.
<http://www.erights.org/elib/concurrency/-event-loop.html>.
- [34] Mark S. Miller, Dean E. Tribble, and Jonathan Shapiro. Concurrency among strangers. In *Trustworthy Global Computing*, pages 195–229, 2005.
- [35] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
- [36] R. Milner. *Lectures on a Calculus for Communicating Systems*, volume 197 of *LNCS*. Springer-Verlag, New York, NY, 1984.
- [37] Robin Milner. *Communicating and Mobile Systems – The Pi Calculus*. Cambridge University Press, June 1999. ISBN:0521658691.
- [38] Gordon E. Moore. *Cramming more components onto integrated circuits*, volume Readings in computer architecture, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
- [39] O. Nierstrasz and M. Papathomas. Viewing objects as patterns of communicating agents. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications European Conference on Object-Oriented Programming (OOPSLA) (ECOOP)*, pages 38–43, Ottawa, ON CDN, [10] 1990. ACM Press , New York, NY , USA.
- [40] Michael Papathomas and Oscar Nierstrasz. Supporting software reuse in concurrent object-oriented languages: Exploring the language design space. Technical report, 1991.
- [41] Michael Philippsen. Imperative concurrent object-oriented languages. Technical Report TR-95-049, Berkeley, CA, 1995.
- [42] B. Garbinato R. Guerraoui and K. Mazouni. Distributed Programming in Garf. In Springer Verlag, editor, *LNCS*, volume 791 of *Object Based Distributed Programming*, pages 225–239, 1994.
- [43] N. R. Scaife. A Survey of Concurrent Object-Oriented Programming Languages. Technical Report RM/96/4, Feb 1996.
- [44] Douglas Schmidt and Steve Vinoski. Programming asynchronous method invocation with CORBA messaging. C++ Report, SIGS, Vol. 11, No 2, February 1999.
- [45] Yuan Shi. Reevaluating amdahl’s law and gustafson’s law. Web Page, October 1996.
<http://www.cis.temple.edu/~shi/docs/amdahl/-amdahl.html>.
- [46] Sun Microsystems. *Java Remote Method Invocation Specification*, 1998.

- [47] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3), March 2005.
- [48] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.
- [49] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 1st edition edition, January 2002. ISBN: 0130888931.
- [50] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, 1998.
- [51] Carlos Varela and Gul Agha. What after Java? from objects to actors. In *Proceedings of the seventh international conference on World Wide Web*, pages p. 573–577, Brisbane, Australia, 1998. Elsevier Science Publishers B. V. Amsterdam, The Netherlands, The Netherlands. ISSN:0169-7552.
- [52] Pierre Vignéras. Mandala. Web page, August 2004.
<http://mandala.sf.net/>.
- [53] Pierre Vignéras. Mandala – SourceForge Project Page. Web page, August 2004.
<http://sourceforge.net/projects/mandala>.
- [54] Pierre Vignéras. Transparency and asynchronous method invocation. In Robert Meersman and Zahir et al. Tari, editors, *On the Move To Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, volume 3760 of *LNCS*, pages 750–762, Agia Napa, Cyprus, 31 October – 4 November 2005. Springer-Verlag. ISBN: 3-540-29736-7.
- [55] Steve Vinoski. New features for CORBA 3.0. *Communications of the ACM*, 41(10):44–52, 1998.