A Language for Multi-threaded Active Objects

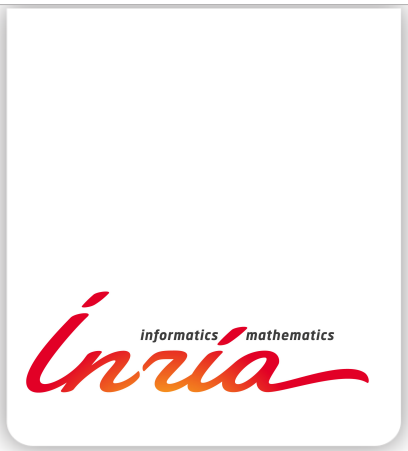
Ludovic Henrio, Fabrice Huet, Zsolt István

A Language for Multi-threaded Active Objects

Ludovic Henrio, Fabrice Huet, Zsolt István

RESEARCH REPORT
N° 8021
July 2012
Project-Team Oasis

ISSN 0249-6399 ISRN INRIA/RR--8021--FR+ENG



A Language for Multi-threaded Active Objects

Ludovic Henrio*, Fabrice Huet*, Zsolt István†

Project-Team Oasis

Research Report n° 8021 — July 2012 — 29 pages

Abstract: The active object programming model is particularly adapted to easily program distributed objects: it separates objects into several *activities*, each manipulated by a single thread, preventing data races. However, this programming model has its limitations in terms of expressiveness – risk of deadlocks – and of efficiency on multicore machines. To overcome these limitations, this paper presents an extension of the active object model, called multi-active objects, that allows each activity to be multithreaded. The new model is implemented as a Java library and relies on method annotations to decide which requests can be run in parallel. It provides implicit parallelism, sparing the programmer from low-level concurrency mechanisms. We define the operational semantics of the multi-active objects and study the basic properties of this model. The benefits of our proposal are highlighted using two different applications: the NAS Parallel Benchmarks and a peer-to-peer overlay.
This report presents the multi-active object programming model, its implementation in Java, its formal semantics and properties, and some experiments showing its efficiency.

Key-words: programming languages, active objects, parallelism, concurrency, distribution

* INRIA-I3S-CNRS, University of Nice Sophia Antipolis, France
† Department of Computer Science, ETH Zurich, Switzerland

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Un Langage pour Objets Actifs Multi-threadés

Résumé : Ce rapport de recherche présente le modèle de programmation à objets multi-actifs. Cette extension des objets actifs est plus adaptée à la programmation des machines multi-cœurs et limite le nombre de dead-locks comparativement à une approche par objets actifs classiques.

Ce rapport décrit le modèle de programmation, son implémentation, sa sémantique, et ses propriétés. Des expérimentations illustrent l'efficacité de l'approche proposée.

Mots-clés : objets actifs, concurrence, distribution

1 Introduction

Programming distributed applications is a difficult task. The distributed application programmer has to face both concurrency issues and location-related issues. The active object [23, 30, 6] paradigm provides a solution for easing the programming of distributed applications by abstracting away the notions of concurrency and of object location. Active objects are similar to actors [2, 1], but better integrated with the notion of objects. The principle of active objects is very simple: an object is said to be active if it acts as an access point to a set of objects and a thread. As a consequence, every call to such an object will be some form of remote method invocation; we call such an invocation a *request*. An active object is thus an object together with a thread that serves the requests it receives.

Active objects are partly inspired by Actors [1]: they share the same asynchronous treatment of messages, and ensure the absence of data race-conditions. They do differ however in how the internal state of the object is represented. One crucial point of active objects (and actors) is that they are mono-threaded and thus prevent data race-conditions without having to use locks or synchronised blocks. In Active objects and in actors, distributed computation relies on absence of sharing between processes allowing them to be placed on different machines.

In classical remote method invocation, the invoker is blocked waiting for the result of the remote call. Active objects, on the other hand, return futures [17] as placeholders for the result, allowing the invoker to continue its execution. Futures can be created and accessed either explicitly, like in Creol [20] or JCoBox [26], or implicitly as in ASP [7] (Asynchronous Sequential Processes) and AmbientTalk [12]. A key benefit of the implicit creation is that no distinction is made between synchronous (i.e., local) and asynchronous (i.e., remote) operations in the program. Hence, when the accessed object is remote, a future is immediately obtained. Similarly to their creation, the access to futures can happen either explicitly using operations like *claim* and *get*, or implicitly, in which case operations that need the real value of an object (*blocking* operations) automatically trigger synchronisation with the future update operation. In most active object languages, future references can be transmitted between remote entities without requiring the future to be resolved. We call those futures that can be transmitted as usual objects *first-class futures*.

After a detailed analysis of existing active object languages in Section 2, this paper presents our contribution which can be summarised as follows.

- A new programming model, called *multi-active objects*[1], is proposed (Section 4). It extends active objects with *local multi-threading*; this both enhances efficiency on multicore machines, and *prevents* most of the *deadlocks* of active objects.
- We rely on declarative *annotations* for expressing potential concurrency between requests, allowing easy and high-level expression of concurrency. However, the expert programmer can still use lower level concurrency constructs like locks.
- We define the *operational semantics* of multi-active objects in Section 5. This semantics allows us to prove properties of the programming model.
- The programming model has been implemented as a Java middleware[2]. We show that *multi-active objects make writing of parallel and distributed applications easier*, but also that the *execution of programs based on multi-active objects is efficient*. Section 6 presents our examples and benchmarks.

The originality of our contribution lies in the interplay between the formal and precise study of the MultiASP language, and a middleware implementation efficient enough to compete with classical multi-threading benchmarks. The use of dynamic constraints for compatibility allows a fine-grain control over local concurrency and improves expressiveness. We also present a simple implementation of a CAN [25] peer-to-peer network in Section 3; it is used both as an illustrative example, and for our benchmarks.

[1] To be precise, the authors published a preliminary and informal description of multi-active object in [19].

[2] available at: `www-sop.inria.fr/oasis/Ludovic.Henrio/java/PA_ma.zip`

```
A a = (A) ProActive.newActive("A", params, Node1); // active object creation
V v = a.bar (...);   // Asynchronous call, no wait, v gets a future
o.gee (v);         // No wait, even if o is a remote active object and v is still awaited
...
v.f (...);        // Wait-by-necessity: wait until v gets its value
```

Figure 1: Typical ProActive code

Section 7 compares our contribution with the closest languages, and finally Section 8 concludes the paper.

2 Overview of Languages for Active Objects

This section overviews the main programming languages that inspired our work. We focus here on active object languages, though other languages such as X10 [8] achieve similar objectives based on different paradigms. Our contribution can be considered as an extension of ASP, but our work shares several points of view with, and can be applied to the other works mentioned in this section.

In Actors [1, 2], distributed computation relies on absence of sharing between processes allowing them to be placed on different machines. Actors interact by asynchronous message passing. They receive messages in their inbox and process them asynchronously. Actors can change their behaviour, whereas active objects have an internal state and make it easy to express stateful behaviours encountered in usual object-oriented languages. Despite the differences in state representation, both programming models ensure the absence of data race-conditions. Several programming languages or models rely on some form of active objects. We review the representative ones below.

Asynchronous Sequential Processes

The ASP calculus [6, 7] is a distributed active object calculus with futures; the ProActive library [5] can be considered as its reference implementation. The ASP calculus formalises the following characteristics of active objects:

- *asynchronous communication*, by a request-reply mechanism;
- *futures*; in ASP, futures are transparent objects: their creation and access is implicit, they are also called *first-class futures*: futures can replace transparently any other objects and can be communicated as the result or parameter of requests;
- *imperative objects*, i.e. each object has its own state;
- *sequential local execution*, each object is manipulated by a single thread.

ASP's active objects ensure the total absence of sharing: objects live in disjoint activities. An activity is a set of objects managed by a unique process and a unique active object. Active objects are accessible through global/distant references. They communicate through asynchronous method calls with futures. ASP is a non-uniform active object model: some of the objects are not active, in which case they are only accessible by a single active object, they are part of its state. Non-uniform active object models are much more efficient because they require fewer messages and fewer threads than uniform active object models.

The code snippet shown in Figure 1 creates a new active object of type `A` on the JVM identified by `Node1`. All calls to that remote object will be asynchronous, and the access to the result might be subject to *wait-by-necessity* (WBN).

The main advantage of ASP is that most code can be written without caring about distribution and concurrency. Futures are automatically and transparently created upon method invocation on an active object. Synchronisation is due to wait-by-necessity that occurs upon access to a future that has not been resolved yet. This synchronisation is performed transparently, i.e. there is no construct for explicitly

waiting the result of a request. Wait-by-necessity is always performed at the last moment, i.e. when a value is really needed.

Futures are also transparently sent between activities. This way simple programs can be written easily and rapidly. Unfortunately, as active objects are purely mono-threaded, ASP's active objects can easily deadlock: most recursive or mutually recursive invocations between active objects lead to a deadlock. In the following, we call this issue *the re-entrance problem*. First-class futures avoid some of those deadlocks when the result of the recursive call is not manipulated, e.g. if it is only returned as a request result.

Several properties have been proved in [6, 7], showing the partial deterministic behaviour of ASP: non-determinism only originates from several requests concurrently sent from two different active objects to the same destination active object. Those properties are ensured because each activity is strictly mono-threaded. Also, some active object programming patterns behave in a process-network [21] manner.

AmbientTalk

In AmbientTalk [12], active objects behave similarly to ASP's active objects. There is one major difference between the two models: in AmbientTalk the future access is a non-blocking operation, it is an asynchronous call that returns another future. There is no wait-by-necessity upon a method call on a future, instead the method call will be performed when the future becomes available. In the meantime another future represents the result of this method invocation. This differs from the approach adopted in other frameworks where access to a future is blocking. This approach avoids the possibility of a deadlock as there is no synchronisation, but programming can become tricky as there is, according to the programming model specification, no way to synchronise two processes.

Creol

Creol [20] is an active object language that executes several requests at the same time, with only one active at a given time. This is some form of *collaborative multi-threading* based on an *await* operation that releases the active thread so that another request can continue its execution. Typically, one would do an *await* when accessing a future so that if the future is not yet available another thread can continue its execution. In Creol [20] future creation and access is explicit, in particular a specific syntax exists for asynchronous method invocation. Creol is a uniform active object model where each object is active and able to receive remote method invocations. Creol also ensures the absence of data races.

De Boer et al. [4] provided the semantics of an object-oriented language based on Creol; it features active objects, asynchronous method calls, and futures. This semantics extends Creol in the sense that it supports first-class futures, although the future access is still explicit (using *get* and *await*). In the same paper, the authors also provide a proof system for proving properties related to concurrency.

The Creol model has the advantage of having less deadlocks than ASP, because in ASP a request must be finished before addressing the next one. Indeed, when the result of a request is necessary in order to finish another one, the Creol programmer can release the service thread, which is impossible in ASP. While no data race condition is possible, interleaving of the different request services triggered by the different release points makes the behaviour more difficult to predict (in particular the determinism properties of ASP cannot be proven in Creol). Recent works [11, 14] allowed the characterisation of deadlocks in Creol but the authors did not provide yet a characterisation of the potential behaviours of the objects, e.g. no analysis of the potential interleaving between request services has yet been proposed.

Overall, explicit future access, explicit release points, and explicit asynchronous calls make the Creol programming model richer than ASP, but also more difficult to program. Finding a good compromise between expressiveness and safe program execution is a crucial aspect in the design of programming languages; we provide here a new extension of the active object model that provides a different tradeoff

between expressiveness and ease of programming compared to the existing approaches. Our new programming model is also more efficient than the existing solutions, at least when running on multicore architectures.

JCoBox

JCoBox [26] is an active object programming model implemented in a language based on Java. It is an active-object language similar to Creol (explicit futures, explicit asynchronous calls towards active objects, collaborative multi-threading, i.e. explicit thread release points). Similarly to Creol, in each cobox a single thread is active at a time, but this thread can be released and coboxes support collaborative multi-threading.

However, it is a non-uniform active-object model that partitions the object space into "coboxes", corresponding to ASP's activities except that a JCobox can contain several active-objects. In ASP and JCoBox, when passed between active objects, passive objects are transmitted by value. References from a cobox to an active object of another cobox are called "far references". Far references can only be used to perform asynchronous calls (`reference!method()`), which return futures. Futures are explicitly created and explicitly accessed, just as in Creol. `await` performs a cooperative wait on the future, whereas `get` blocks until the value of the future is received. In JCoBox, contrary to ASP, a cobox may contain multiple active objects. In this paper, we have a single active object per activity; the resulting model is simpler, but extending our model to multiple active objects per activity is quite easy and could add, in some cases, expressiveness to the programming model.

Figure 2 shows explicit future creation, and explicit future accesses in JCoBox. When inside an active object a single thread is active at a time, accessing futures can lead to deadlock in case of re-entrant requests. The solution proposed by ASP and ProActive is "first-class futures": since futures are implicitly created and transparently transmitted as method parameters and results, the deadlock only occurs if the future is really needed. Alternatively, Creol and JCoBox provide explicit futures and allow the active thread to suspend itself until a result is returned. Consider the method `foo` of Figure 2, if one replaces the `await` statement by a `get`, the active object would deadlock waiting for `bar` to be executed. It might seem a safe programming guideline to systematically perform an `await` instead of a `get`. However, this might lead to unexpected non-determinism or unexpected results. In JCoBox, ready threads (i.e. threads that can execute but are suspended) are dequeued in a FIFO order. In the example, the `bar` and the second `foo` request will then be executed in an unpredictable order. Depending on when the second `foo` is received, the final result returned by the first `foo` may be '2'. This happens because 'x' can be modified by the second `foo` request before the return statement of the first.

In this paper, we propose an alternative approach where concurrency is possible, no explicit control on futures is necessary, and the programmer can choose which request can be executed at the same time as another one. A program equivalent to Figure 2 can be written in our framework, and provided the programmer states that `foo` can execute in parallel with `bar` but not with another `foo` request, the result would always be the expected one: '1 2'.

JAC

JAC [18] is an extension of Java that introduces a higher level of concurrency and separates thread synchronisation from application logic in an expressive and declarative fashion. JAC relies on a custom precompiler and declarative annotations, in the form of Javadoc comments placed before method headers or inside the code. Objects are annotated as `controlled` when their threads are managed and synchronised according to JAC's annotations. JAC relies on compatibility annotations stating whether two methods can be executed at the same time; two methods should be compatible if they do not access the same variables

```
@CoBox class SomeClass { //declaring a cobox
  int x;

  int bar() {
    return 0;
  }

  //sets value of x, but may release the thread
  int foo(int v) {
    x=v;
    Fut<int> z=this!bar(); //async. call on itself
    ...
    int res=z.await(); // allows another request to progress
    return x+res;
  }
}

//in some other class
SomeClass s = ...
a = s!foo(1);
b = s!foo(2);
print(a.get()+' '+b.get());
//the output could be either '1 2' or '2 2'!
```

Figure 2: Thread interleaving in JCoBox may lead to unexpected outcomes

(or if the access to those variables has been protected by some locking mechanism). For example, the following code states that `isEmpty` can be safely executed concurrently with `lookup` and with itself:

```
/** @compatible lookup(Object), isEmpty() */
  public boolean isEmpty() {.....}
```

Additional annotations are given for finer grained or easier control of concurrency and synchronisation (e.g. to wait for a guard to be verified before executing a method). [18] also presents an exhaustive case study of the annotations, in particular in relation with inheritance. JAC's *async* annotation provides some form of active object behaviour: an asynchronous method is executed independently of others in a separate thread. The main difference with classical active objects is that classical active objects act as a unit of concurrency: they are manipulated with a single thread and enforce the absence of shared memory between active objects. Uniform active-objects could be obtained in JAC by stating that all methods of all classes are asynchronous and mutually exclusive, but JAC cannot be used to ensure the absence of sharing featured by non-uniform active object languages. This makes JAC less adapted to the programming of distributed objects. It does not provide data and concurrency abstraction ensured by classical active-objects.

We think JAC is a well designed model for declaring powerful concurrency rules in a simple manner. Unfortunately it is neither particularly adapted to a distributed environment, nor does it compare to active objects in terms of encapsulation of data and concurrency. In this paper we try to overcome the limitations of the two worlds and provide a programming model featuring both the distribution support from active objects and the local declarative concurrency from JAC. At the same time, compared to JAC, we also extend the expressiveness of declarative concurrency by defining runtime-compatibility rules.

In this section we reviewed active object languages and their limitations. ASP comes with strong properties and is to our mind the easiest model to program, but the fact that a request must be served without interruption might create a lot of deadlocks. Even though JCoBox and Creol provide an elegant solution for data-space partitioning and solve the re-entrance problem mentioned earlier, they rely on explicitness which makes the writing of simple applications quite complex and does not allow precise control over local concurrency. Additionally, none of the active-object frameworks allows real multi-threaded execution, which is a drawback on multi-core machines.

Overall, the frameworks presented above provide nice abstractions for concurrency, but are sometimes complex, or do not support distribution. While providing a programming model that is easy to program, multi-active objects feature both data-encapsulation inside an activity, and local concurrency. Moreover, while data-races are not strictly avoided, we provide a strong support to avoid or control parallelism. Last but not least, multi-active objects support multi-threaded execution, making them efficient on multi-core machines.

3 Illustrative Example: CAN

We will illustrate our proposal with an example inspired by a simple peer-to-peer network based on the CAN [25] routing protocol. In a CAN, data are stored in peers as key-value pairs inside a local datastore. Keys are mapped through a bijective function to the coordinates of a N-dimensionary space, called the key-space. The key-space is partitioned so that each key is owned by a single peer. A peer knows its immediate neighbours, and when an action concerns a key that does not belong to this particular peer, it routes the request towards the responsible peer according to the coordinates of the key. Figures 3 and 4 illustrate the CAN routing algorithm for the two scenarios that we study in Section 6.3.

Our CAN example provides three operations: *join*, *add*, and *lookup*. When a new peer *joins* another one already in the network, the joined peer splits its key-space and transfers the associated data to the joining peer. The *add* operation stores a key-value pair in the network, and *lookup* retrieves it. CAN provides other operations which are not particularly interesting for this paper and are thus omitted here.

Active objects are great for harnessing parallelism that originates from distribution, thus it is a natural choice to implement this application by representing each peer with an active object. A consequence of this choice is that, locally, requests will be served one-by-one. This on one hand limits the performance of the application, and on the other hand possibly leads to deadlocks if re-entrant requests are issued (for instance a *lookup* that is accidentally routed through the same peer twice). With multi-active objects, apart from benefiting from the inherent distributed parallelism of the system, a peer will be able to handle several operations in parallel. In the following section we will illustrate the fitness of multi-active objects by showing how to simply program a safe parallelisation of the CAN peers.

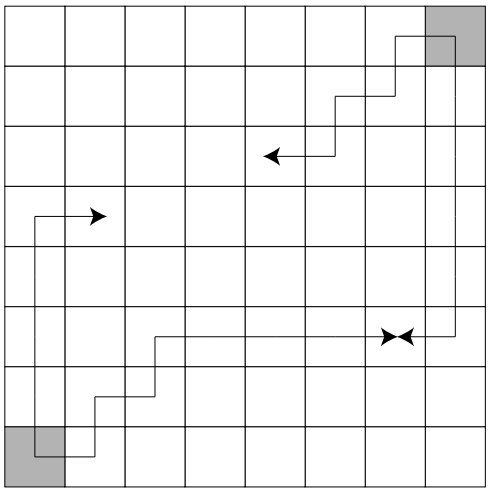


Figure 3: CAN routing from two corners

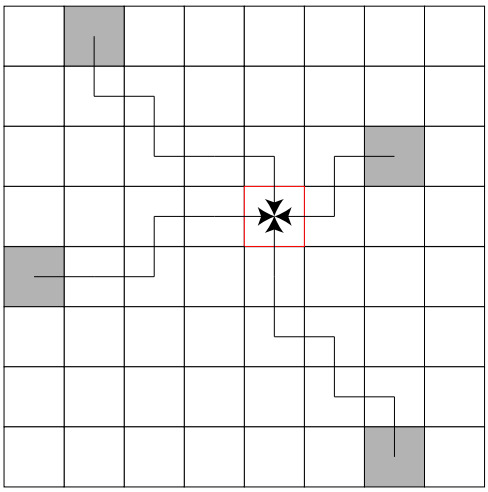


Figure 4: All nodes accessing centre

4 Multi-active Objects

4.1 Assumptions and Design Choices

To overcome the limitations of active objects with regard to parallel serving of requests, we introduce multi-active objects that enable local parallelism inside active objects in a safe manner. For this the programmer can annotate the code with information about concurrency by defining a compatibility relationship between concerns. For instance, in our CAN example we distinguish two non-overlapping concerns: one is network management (*join*) and another is routing (*add*, *lookup*). For two concerns that deal with completely disjoint resources it is possible to execute them in parallel, but for others that could conflict on resources (e.g. joining nodes and routing at the same time in the same peer) this must not happen. Some of the concerns enable parallel execution of their operations (looking up values in parallel in the system will not lead to conflicts), and others do not (a peer can split its zone only with one other peer at a time).

In the RMI style of programming, every exposed operation of an object will be run in parallel, thus methods belonging to different concerns could execute at the same time inside an object. A classic approach to solve this problem is to protect all the data accesses, by using *synchronized* blocks inside methods. While this approach works well when most of the methods are incompatible with each other, this all-or-nothing behaviour becomes inefficient in case there is a more complex relationship between methods.

By nature, active objects materialise a much safer model where no inner concurrency is possible. We extend this model by assigning methods to *group*s (concerns). Then, methods belonging to compatible groups can be executed in parallel, and methods belonging to conflicting groups will be guaranteed not to be run concurrently. This way the application logic does not need to be mixed with low-level synchronisations. The idea is that *two groups should be made compatible if their methods do not access the same data, or if concurrent accesses are protected by the programmer,* and *the two methods can be executed in any order*. Overall, the programmer has the choice of either setting compatibility between only non-conflicting groups in a simple manner, or protecting the conflicting code by means of locks or synchronised blocks for the most complex cases.

In this work, we assume that the programmer defines groups and their compatibility relations inside a class in a safe manner. Of course dynamic checks or static analysis should be added to ensure, for example, that no race condition appear at runtime. However, we consider such an extension to the framework deserves a separate study, and decide to focus in this paper on the programming model itself.

We start from active objects *à la* ASP, featuring transparent creation, synchronisation, and transmission of futures. We think that the transparency featured by ASP and ProActive helps writing simple programs, and is not an issue when writing complex distributed applications. However, this choice is not crucial here, and the principles of multi-active objects could as well be applied to an active object language with explicit futures. We also think that ASP, like JCoBox, features non-uniform active objects that reflects better the way efficient distributed applications are designed: some objects are co-allocated and only some of them are remotely accessible. In our current implementation and model, only one object is active in a given activity (or cobox) but our model could easily be extended to multiple active-object per cobox.

An active object can be transformed into a multi-active object by applying the following design methodology. Without annotations, a multi-active object behaves similarly to an active object, no race condition is possible, but no local parallelism is possible. If some parallelism is desired, e.g. for efficiency reason or because dead-locks appeared, each remotely invocable method can be assigned to a group and two groups are declared as compatible if no method of one group accesses the same variable as a method of another group. In that case, method of the two groups will be executed in parallel and without respecting the original order, meaning the groups can only be declared compatible if additionally

```
@DefineGroups({
  @Group(name="join", selfCompatible=false)
  @Group(name="routing", selfCompatible=true)
  @Group(name="monitoring", selfCompatible=true)
})
@DefineRules({
  @Compatible({"join", "monitoring"})
  @Compatible({"routing", "monitoring"})
})
public class Peer {
@MemberOf("join")
  public JoinResponse join(Peer other) { ... }
@MemberOf("routing")
  public void add(Key k, Serializable value) { ... }
@MemberOf("routing")
  public Serializable lookup(Key k) { ... }
@MemberOf("monitoring")
  public void monitor() { ... }
}
```

Figure 5: The CAN Peer annotated for parallelism

the order of execution of method of one group relatively to the other is not significant. If more parallelism is still required, the programmer has two non-exclusive options: either he protects the access to some of the variables by a locking mechanism which will allow him to declare more groups as compatible, or he realises that, depending on runtime conditions (invocation parameters, object's state, ...) some groups might become compatible and he defines a compatibility function allowing him to decide at runtime which request executions are compatible.

We now describe in details the multi-active objects framework we designed and implemented.

4.2 Defining Groups

The programmer can use an annotation (`Group`) to define a group and can specify whether the group is `selfCompatible`, i.e., two requests on methods of the group can run in parallel. The syntax for defining groups in the class header is:

```
@DefineGroups({
  @Group(name="uniqueName" [, selfCompatible=true|false])
[, ...] })
```

Compatibilities between groups can be expressed as `Compatible` annotations. Each such annotation receives a set of groups that are pairwise compatible:

```
@DefineRules({
  @Compatible({"groupOne", "groupTwo", ...})
[, ...] })
```

Finally, a method's membership to a group is expressed by annotating the method's declaration with `MemberOf`. Each method belongs to only one group. In case no membership annotation is specified, the method belongs to an anonymous group that is neither compatible with other groups, nor self-compatible. This way, if no method of a class is annotated, the multi-active object behaves like an ordinary active object.

```
@MemberOf("nameOfGroup")
```

```
@DefineRules({
  @Compatible({"groupOne", "groupTwo", ...})
[, ...] })
```

```
@DefineGroups({
  @Group(name="routing", selfCompatible=true, parameter="can.Key"),
  @Group(name="join", selfCompatible=false) })
@DefineRules({
  @Compatible(value={"routing", "join"},condition="!this.isLocal") })
public class Peer {
  @MemberOf("join")
  public JoinResponse join(Peer other) {
    ... //split the zone of the peer (into 'myNewZone' and 'otherZone')
    ... //create response for the joining peers with 'otherZone' and its data
    synchronized (lock) { myZone = myNewZone; }
    return response;
  }
  @MemberOf("routing")
  public void add(Key k, Serializable value) { ... }

  private boolean isLocal(Key k){
    synchronized (lock) { return myZone.containsKey(k); }
  }
}
```

Figure 6: The CAN Peer annotated for parallelism with dynamic compatibility

Figure 5 shows how these annotations are used in the Java class implementing a CAN peer in which *add*s and *lookup*s can be performed in parallel – they belong to the same self-compatible group `routing`. Since there is no compatibility rule defined between the groups, methods of `join` and `routing` will not be served in parallel. To fully illustrate our annotations, we added *monitoring* as a third concern independent from the others as illustrated in the example.

We chose annotations as a vehicle for parallel compatibility definitions because these are strongly dependent on the application logic, and in our opinion, should be written at the same time as the application. The related work of Shanneb et. al [28] considers the case when not only the interface of a composite object (in our case the active object), but also its composing objects are annotated. While this technique would make it possible to infer the parallel behaviour of the active object based on annotated passive objects contained in it, we argue that for most real-world applications setting the rules up at the active object level is simpler and expressive enough.

4.3 Dynamic Compatibility

Sometimes it is desirable to decide the compatibility of some requests at run-time, depending on the state of the active object, or the parameters of the requests. For example, two methods writing in the same array can be compatible if they don't access the same cells.

For this reason, we first introduce an optional *group-parameter* which indicates the type of a a parameter which will be used to decide compatibility. This parameter must appear in all methods of the group and in case a method has several parameters of this type, the leftmost one is chosen. In Figure 6, we add `parameter="can.Key"` to the *routing* group to indicate that the parameter of type *Key* will be used. To actually decide the compatibility, we add a condition in the form of a *compatibility function*. This function takes as input the common parameters of the two compared groups and returns *true* or *0* if the methods are compatibles. The general syntax for a dynamic compatibility rule is:

```
  @compatible{value={"group1","group1"}, condition="SomeCondition"}
```

The compatibility function can be defined as follows:

- when `SomeCondition` is in the form `someFunc`, the compatibility will be decided by executing `param1 .someFunc(param2)` where `param1` is the parameter of one request and `param2` is the parameter of the other.

- when `SomeCondition` is in the form `[REF].someFunc`, the compatibility will depend on the results of `someFunc(param1, param2)` with the group parameters as arguments. `[REF]` can be either **`this`** if the method belongs to the multi-active object itself, or a class name if it is a static method.

Additionally the result of the comparator function can be negated using "!", e.g. `condition="!this.isLocal"`. If there are several orders of group parameters for which the compatibility function exists then the order is unspecified and any function of the right name and signature is called, this is why, generally, the compatibility method should be symmetrical. Since the compatibility method can run concurrently with executing threads, it is the responsability of the programmer to ensure mutual exclusion, if necessary.

One can define dynamic compatibility when only one of the two groups has a parameter, in that case the compatibility function should accept one input parameter less. It is even possible to decide dynamically compatibility when none of the two groups has a parameter (e.g. based on the state of the active object); in that case the compatibility function should be a static method or a method of the active object, with no parameter.

As an example, we show below how to better parallelise the execution of joins and routing operations in our CAN. During a *join* operation, the peer which is already in the network splits its key-space and transfers some of the key-value pairs to the peer which is joining the network. During this operation, ownership is hard to define. Thus a *lookup* (or *add*) of a key belonging to one of the two peers cannot be answered during the transition period. Operations that target "external" keys, on the other hand, could be safely executed in parallel with a join. Figure 6 shows how to modify the peer with dynamic compatibility checks:

- The function `isLocal` checks whether a key belongs to the zone of the peer. This method relies on a **`synchronized`** statement to ensure that the threads running the application logic and the ones evaluating the request compatibilities will not conflict.
- The key, the common parameter of `add` and `lookup`, was added as a parameter to the group of routing operations.
- A compatibility rule was added that allows `join`s and the `routing` operations to run in parallel in case the key of these operations is not situated in the zone of the peer.

Note that since we did not define a condition for self-compatibility, the parallel routing behaviour remains unchanged. However, if we would want to guarantee that there is no overtaking between `routing` requests on the same key, then it is sufficient to state that the group `routing` is `selfcompatible` only when the key parameter of the two invocations is not equal, which is declared as follows:

```
@Group(name="routing",selfCompatible=true,parameter="can.Key",condition="!equals")
```

4.4 Scheduling Request Services

In active-object languages, requests are served in a FIFO order if no particular service policy is specified. In multi-active objects, even though we focus on increasing the parallelism inside active objects, we also provide guarantees on the order of execution. Section 5.3 will define a service policy that maximises parallelism while ensuring that only compatible requests are served in parallel. We achieve this by serving the first request that is compatible with all the requests currently served and all the requests preceding it in the queue.

Explanations on the opimal request policy A naive policy would serve the requests in the order that they arrive, and provided that the first request in the queue is compatible with the currently served ones, it will be served in parallel with them. However, this solution does not always ensure maximum parallelism inside multi-active objects. Consider the following example. Inside a CAN peer there are two concurrent *add* operations running and there are two requests in the queue: *join* and *monitor* (member of the group `monitoring`). Using the FIFO logic presented above, we would not be able to start any more requests until

```
public List<Request> runPolicy(StatefulCompatibilityMap compatibility) {
  List<Request> ret = new LinkedList<Request>();
  Request oldest = compatibility.getOldestInTheQueue();
  if (compatibility.isCompatibleWithExecuting(oldest) && (isNotAdd(oldest) || countAdds(
       compatibility.getExecutingRequests())<N)) {
    ret.add(oldest);
  }
  return ret;
}
```

Figure 7: Custom scheduling policy limiting the number of concurrent *add*s

the two *adds* finish. However, the *monitor* request is compatible with all the others, and it could be safely executed before *join*, concurrently with *adds*. This leads us to an optimisation of the serving policy: a request can be served in parallel if *it is compatible with all running requests and all the requests preceding it in the queue*. This policy ensures maximum parallelism while maintaining the relative ordering of non-compatible requests. Section 5.4 provides further comparison between these two policies.

Customizing policies Besides the built-in policy, multi-active objects expose an API that lets programmers plug in their custom policies for scheduling requests. A scheduling policy is a function that takes as input 1) the set of running requests, 2) the contents of the queue, and 3) the compatibility information given in the class. It returns a set of requests that can be started right away. During the execution of the scheduling policy, the set of running methods, and the compatibility information are guaranteed not to change, thus access to those information does not need to be protected by locks, only access to the applicative state of the active object might require locks. The scheduling policy is run every time there is a change in the set of running or waiting requests.

As an illustration of user-defined scheduling policies, consider the CAN example and suppose that a peer can only store N values in parallel into the datastore. Figure 7 shows a scheduling policy that implements a FIFO policy but limits the number of parallel *add*s to N. Note that in the actual API, the three inputs mentionned in the previous paragraph are wrapped in a `StatefulCompatibilityMap` object, and the policy may return several requests. For the sake of this simple example we return requests one by one. An equivalent method returning all the requests ready for execution could also be implemented. The methods `isNotAdd` and `countAdds` are user-defined.

4.5 Inheritance

Sub-classing is a basic building block of object oriented applications, and it clearly would be infeasible to re-declare compatibility information every time a class is extended. Therefore we designed our annotations to have an inheritance behaviour similar to Java classes: implicitly, parallel behaviour is inherited with the logic, but the programmer can add or override definitions in the subclass if necessary.

More precisely, groups defined in a class will persist throughout all of its subclasses, and may not be re-declared. However, subclasses may define new groups. The membership of a method is inherited, unless the method is overridden. In this case the membership has to be re-declared as well. When overriding methods in subclasses, their membership can be set to any group defined in the class or the super classes; but it can also be omitted, resulting in mutual exclusion with everyone else. Compatibility functions can be overridden in subclasses, but it is reasonable to declare compatibility functions *final* as their correctness strongly depends on the exact behaviour of the served requests, and overriding these compatibility functions allows a sub-class to change the compatibility between existing groups.

4.6 Limiting the Number of Threads

By default, multi-active objects serve as many compatible requests in parallel as possible. In practice, while applications mostly benefit from running on several concurrent threads, the performance could degrade if too many threads are created on a single machine. To overcome this issue in our implementation we introduce a two-level thread limitation mechanism, that we implemented separately from the scheduling policy. This way, scheduling policies focus on the logical aspects, while the thread limitation strategies can ensure that the execution fits the hardware capabilities of the underlying machine. There are two levels of thread limitation:

Active thread limit restricts the number of active threads inside a multi-active object. A request (and its thread) is said to be active if it is being served and is not performing a wait-by-necessity (WBN). This limitation mechanism has the advantage that it cannot induce deadlocks caused by re-entrance (when a request performs a WBN on the result of an enqueued request), but it is still possible to end up with a high number of threads.

Strict limit restricts the total number of threads (active or in WBN) inside the multi-active object. While this solution always prevents too many threads from being created it may deadlock because of re-entrance if the multi-active object runs out of threads to serve requests needed to release the WBN.

5 Semantics

This section describes MultiASP, the multi-active object calculus. We present its small step operational semantics and its properties. In MultiASP, there is no explicit notion of place of execution, but the calculus is particularly adapted to distribution because first, inter-activity communications behave like remote method invocation, and second each object belongs to a single activity. Overall each active object can be considered as a unit of distribution. The operational semantics is parameterised by a function deciding whether a request should be served concurrently on a new thread or sequentially by the thread that triggered the service.

5.1 Syntax and Runtime Structures

5.1.1 Static Terms

While x, y range over variable names, we let l_i range over field names, and m_i over method names (m_0 is a reserved method name; it is called upon the activation of an object and encodes the service policy). In this section, :: denotes the concatenation of lists; we also use it to append an element to a list. $\varnothing$ denotes an empty list or an empty set. The set P of programs (static terms) is the same for ASP [7] and MultiASP[3], note that every program is an object and that [] is an empty object:

$$
\begin{array}{lr}
P ::= x & \text{variable,} \\
\quad | \, [l_i = P_i; m_j = \varsigma(x_j, y_j) P'_j]^{i\in 1..n}_{j\in 1..m} & \text{object definition,} \\
\quad | \, P.l_i & \text{field access,} \\
\quad | \, P.l_i := P' & \text{field update,} \\
\quad | \, P.m_j(P') & \text{method call,} \\
\quad | Active(P) & \text{activates } P, \\
\quad | \mathit{Serve}(M) & \text{serves a request among } M, \\
 & \text{a set of method labels.} \\
 & M = \{m_i\}^{i\in 1..N}
\end{array}
$$

Note that each method definition (inside the object definition) has two parameters: one binds the object itself (self variable like in ς-calculus), the other binds the actual method parameter.

[3] We removed the clone operator from ς-calculus; it is not particularly interesting here.

5.1.2 Runtime Terms

At runtime new objects can be allocated in a local store (there is one local store for each active object), and new active objects and futures can be created. Thus, we let ι_i range over references to the local store (ι_0 is a reserved location for the active object), α, β, γ range over active object identifiers, and f_i range over future identifiers. Compared to static terms, runtime terms (a_i, b_i) can also contain references to futures, active objects, and to store location:

$$\begin{array}{lll} a_i ::= x \mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n} \mid a.l_i & & \\ \quad \mid a.l_i := b \mid a.m_j(b) \mid \mathit{Active}(a) \mid \mathit{Serve}(M) & & \\ \quad \mid \iota & & \text{location} \\ \quad \mid f & & \text{future reference} \\ \quad \mid \alpha & & \text{active object reference} \end{array}$$

5.1.3 Structure of Activities

A *reduced object* is either a future, an activity reference, or an object with all fields reduced to a location. A store maps locations to reduced objects; it stores the *local state* of the active object:

$$o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n} \mid f \mid \alpha$$

A store maps locations to reduced objects, and it is used to store the *local state* of the active object:

$$\sigma ::= \{\iota_i \mapsto o_i\}^{i\in 1..k}$$

A substitution is denoted by $a\{b \leftarrow c\}$ for replacing (free) occurrences of b by c in a; we will apply it indifferently to variables or locations.

We define the following operations on store. First, $\{\iota \mapsto o\} + \sigma$ returns a store similar to σ except the entry for ι that is now associated to o; Second, to ensure absence of sharing, the operation *Copy&Merge*$(\sigma, \iota\,;\, \sigma', \iota')$ performs a *deep copy* of the object at location ι in σ into the location ι' of σ', and ensures that communicated values are "self-contained". This is achieved by copying the entry for ι at location ι' of σ' (if this location already contains an object it will be erased). All locations ι'' referenced (recursively) by the object $\sigma(\iota)$ are also copied in σ' at fresh locations This operators is defined formally as follows:

$$\begin{array}{c} \mathit{Copy\&Merge}(\sigma, \iota\,;\, \sigma', \iota') \triangleq \sigma' + \mathrm{copy}(\iota, \sigma)\{\iota \leftarrow \iota'\}\theta \\ \text{where } \theta = \{\iota_1 \leftarrow \iota_2 \mid \iota_1 \in \mathit{dom}(\mathrm{copy}(\iota, \sigma))\backslash\{\iota'\}, \iota_2 \text{ fresh}\} \end{array}$$

copy is an auxiliary operator that captures the part of the store σ starting at location ι, it returns an independent partial store ($\iota'' \in \sigma(\iota')$ is a shortcut for "ι'' is a location appearing in the object stored at location ι of σ).

$$\iota \in \mathit{dom}(\mathrm{copy}(\iota, \sigma))$$

$$\iota' \in \mathit{dom}(\mathrm{copy}(\iota, \sigma)) \wedge \iota'' \in \sigma(\iota') \Rightarrow \iota'' \subseteq \mathit{dom}(\mathrm{copy}(\iota, \sigma))$$

$$\iota' \in \mathit{dom}(\mathrm{copy}(\iota, \sigma)) \Rightarrow \mathrm{copy}(\iota, \sigma)(\iota') = \sigma(\iota')$$

From this, *Copy&Merge*$(\sigma, \iota\,;\, \sigma', \iota')$ is formed by appending σ' and $\mathrm{copy}(\iota, \sigma)$ replacing ι by ι' and renaming the other locations so that no conflict occurs.

F ranges over *future value association lists*; such a list stores computed results where ι_i is the location of the value associated with the future f_i: $F ::= \{f_i \mapsto \iota_i\}^{i\in 1..k}$. The list of *pending requests* is denoted by $R ::= [m_i; \iota_i; f_i]^{i\in 1..N}$, where each request consists of: the name of the *target method* m_i, the location of the *argument* passed to the request ι_i, the *future* identifier which will be associated to the result f_i. $(f \mapsto \iota) \in F$ means $(f \mapsto \iota)$ is one of the entries of the list F and similarly $[m; \iota; f] \in R$ means $[m; \iota; f]$ is one of the requests of the queue R.

STORE
$$\frac{\iota \notin dom(\sigma)}{(\mathcal{R}[o],\sigma) \rightarrow_{\text{loc}} (\mathcal{R}[\iota],\{\iota \mapsto o\}+\sigma)}$$

FIELD
$$\frac{\sigma(\iota)=[l_i=\iota_i; m_j=\varsigma(x_j,y_j)a_j]^{i\in 1..n}_{j\in 1..m} \qquad k\in 1..n}{(\mathcal{R}[\iota.l_k],\sigma) \rightarrow_{\text{loc}} (\mathcal{R}[\iota_k],\sigma)}$$

INVOKE
$$\frac{\sigma(\iota)=[l_i=\iota_i; m_j=\varsigma(x_j,y_j)a_j]^{i\in 1..n}_{j\in 1..m} \qquad k\in 1..m}{(\mathcal{R}[\iota.m_k(\iota')],\sigma) \rightarrow_{\text{loc}} (\mathcal{R}[a_k\{x_k \leftarrow \iota, y_k \leftarrow \iota'\}],\sigma)}$$

UPDATE
$$\frac{\sigma(\iota)=[l_i=\iota_i; m_j=\varsigma(x_j,y_j)a_j]^{i\in 1..n}_{j\in 1..m} \qquad k\in 1..n \qquad o'=[l_i=\iota_i; l_k=\iota'; l_{k'}=\iota_{k'}; m_j=\varsigma(x_j,y_j)a_j]^{i\in 1..k-1,k'\in k+1..n}_{j\in 1..m}}{(\mathcal{R}[\iota.l_k := \iota'],\sigma) \rightarrow_{\text{loc}} (\mathcal{R}[\iota],\{\iota \mapsto o'\}+\sigma)}$$

Table 1: Local reduction

We have two parallel composition operators: $\|$ is a *local parallelism operator* separating threads residing in the same activity, and $|||$ is the *distributed parallelism operator* separating two active objects or two locations.

A request being evaluated is a term together with the future to which it corresponds ($a_i \mapsto f_i$). An activity has several parallel threads each consisting of a list of request being treated: the first request of each thread is in fact currently being treated, the others are in a waiting state. C is a *current request structure*: it is a parallel composition of threads where each thread is a list of requests. By nature, the $\|$ operator is symmetric, and current requests are identified modulo reordering of threads:

$$C ::= \varnothing \mid [a_i \mapsto f_i]^{i\in 1..n} \| C$$

Finally, an activity is composed of a name α, a store σ, a list of pending requests R, a set of computed futures F, and a current request structure C. A configuration Q is made of activities. Configurations are identified modulo the reordering of activities.

$$Q ::= \varnothing \mid \alpha[F; C; R; \sigma] ||| Q$$

To transform a static term P_0 into a configuration that can be evaluated, we create an *initial configuration*, denoted Q_0, consisting in putting the program as the current request of an activity (associated with a fresh unreachable future $f_\varnothing$): $Q_0 = \alpha_0[\varnothing; P_0 \mapsto f_\varnothing; \varnothing; \varnothing]$.

5.1.4 Contexts

Reduction contexts are terms with a hole indicating where the reduction should happen. For each context $\mathcal{R}$, an operation ($\mathcal{R}[c]$) is defined for filling the hole, this operation replaces the hole by a given term c: $\mathcal{R}[c] = \mathcal{R}\{\bullet \leftarrow c\}$. Contrarily to substitution, filling a hole is not capture avoiding: the term filling the hole is substituted as it is.

Sequential reduction contexts indicate where reduction occurs in a current request:

$$\begin{aligned}\mathcal{R} ::= {} & \bullet \mid \mathcal{R}.l_i \mid \mathcal{R}.m_j(b) \mid \iota.m_j(\mathcal{R}) \mid \mathcal{R}.l_i := b \mid \iota.l_i := \mathcal{R} \\ & \mid [l_i=\iota_i, l_k=\mathcal{R}, l_{k'}=b_{k'}; m_j=\varsigma(x_j,y_j)a_j]^{i\in 1..k-1,k'\in k+1..n}_{j\in 1..m} \\ & \mid Active(\mathcal{R})\end{aligned}$$

A *parallel reduction context* extracts one thread of the current request structure (remember that current requests are identified modulo thread reordering):

$$\mathcal{R}_c ::= \quad [\mathcal{R} \mapsto f_1] :: [a_j \mapsto f_j]^{j\in 2..n} \quad \| C$$

LOCAL
$$\frac{(a,\sigma) \rightarrow_{\text{loc}} (a',\sigma')}{\alpha\big[F;\mathcal{R}_c[a];R,\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;\mathcal{R}_c[a'];R;\sigma'\big] \,|||\, Q}$$

ACTIVE
$$\frac{\gamma \text{ fresh activity name} \qquad f_\varnothing \text{ fresh future} \qquad \sigma_\gamma = \mathit{Copy\&Merge}(\sigma,\iota\,;\,\varnothing,\iota_0)}{\alpha\big[F;\mathcal{R}_c[\mathit{Active}(\iota)];R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;\mathcal{R}_c[\gamma];R;\sigma\big] \,|||\, \gamma\big[\varnothing;[\iota_0.m_0([]) \mapsto f_\varnothing];\varnothing;\sigma_\gamma\big] \,|||\, Q}$$

REQUEST
$$\frac{\sigma_\alpha(\iota) = \beta \qquad \iota'' \notin \mathit{dom}(\sigma_\beta) \qquad f \text{ fresh future} \qquad \sigma'_\beta = \mathit{Copy\&Merge}(\sigma_\alpha,\iota'\,;\,\sigma_\beta,\iota'')}{\alpha\big[F;\mathcal{R}_c[\iota.m_j(\iota')];R;\sigma_\alpha\big] \,|||\, \beta\big[F';C';R';\sigma_\beta\big] \,|||\, Q \longrightarrow \alpha\big[F;\mathcal{R}_c[f];R;\sigma_\alpha\big] \,|||\, \beta\big[F';C';R'::[m_j;\iota'';f];\sigma'_\beta\big] \,|||\, Q}$$

ENDSERVICE
$$\frac{\iota' \notin \mathit{dom}(\sigma) \qquad \sigma' = \mathit{Copy\&Merge}(\sigma,\iota\,;\,\sigma,\iota')}{\alpha\big[F;\iota\mapsto f::[a_i\mapsto f_i]^{i\in 1..n} \| C;R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F::f\mapsto\iota';[a_i\mapsto f_i]^{i\in 1..n} \| C;R;\sigma'\big] \,|||\, Q}$$

REPLY
$$\frac{\sigma_\alpha(\iota) = f \qquad \sigma'_\alpha = \mathit{Copy\&Merge}(\sigma_\beta,\iota_f\,;\,\sigma_\alpha,\iota) \qquad (f\mapsto\iota_f) \in F'}{\alpha\big[F;C;R;\sigma_\alpha\big] \,|||\, \beta\big[F';C';R';\sigma_\beta\big] \,|||\, Q \longrightarrow \alpha\big[F;C;R;\sigma'_\alpha\big] \,|||\, \beta\big[F';C';R';\sigma_\beta\big] \,|||\, Q}$$

SERVE
$$\frac{C = [\mathcal{R}[\mathit{Serve}(M)]\mapsto f_0]::[a_i\mapsto f_i]^{i\in 1..n} \| C' \qquad \mathit{SeqSchedule}(M,\{f_i\}^{i\in 0..n},\mathit{Futures}(C'),R) = ([m,f,\iota],R')}{\alpha\big[F;C;R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;[\iota_0.m(\iota)\mapsto f]::[\mathcal{R}[[]]\mapsto f_0]::[a_i\mapsto f_i]^{i\in 1..n} \| C';R';\sigma\big] \,|||\, Q}$$

PARSERVE
$$\frac{C = [\mathcal{R}[\mathit{Serve}(M)]\mapsto f_0]::[a_i\mapsto f_i]^{i\in 1..n} \| C' \qquad \mathit{ParSchedule}(M,\{f_i\}^{i\in 0..n},\mathit{Futures}(C'),R) = ([m,f,\iota],R')}{\alpha\big[F;C;R;\sigma\big] \,|||\, Q \longrightarrow \alpha\big[F;[\iota_0.m(\iota)\mapsto f] \| \mathcal{R}[[]]\mapsto f_0]::[a_i\mapsto f_i]^{i\in 1..n} \| C';R';\sigma\big] \,|||\, Q}$$

Table 2: Parallel reduction (used or modified values are non-gray)

5.2 Semantics

Our semantics is built in two layers, a local reduction $\rightarrow_{\text{loc}}$ that corresponds to a classical object calculus, and a parallel semantics that encodes distribution and communications.

The local semantics (Table 1) is very similar to ASP [7] except that it does not use reduction context because they are already used in the distributed semantics. It also shares a lot of similarities with with the semantics of **imp**ς-calculus [15]. As shown in Table 1; local reduction consists of one rule for storing a reduced object[4], one performing a field access, one invoking a method (note here that, classically, substitution is capture avoiding), and one updating a field.

From the local layer that corresponds to a classical object calculus, we build a semantics for multi-active objects that encodes distribution and communications. Activities communicate by remote method invocations and handle several local threads; each thread can evolve and modify the local store according to the local reduction rules. Thanks to the parallel reduction contexts $\mathcal{R}_c$, multiple threads are handled almost transparently in the semantics. The main novelty in MultiASP is the request service that can either serve the new request in the current thread or in a concurrent one; for this we rely on two methods *SeqSchedule* and *ParSchedule* with the same signature:

[4]remember o can be a future or an activity reference

$$\text{SELFREQ}\quad \frac{\sigma_\alpha(\iota)=\alpha \qquad \iota'' \notin dom(\sigma_\alpha) \qquad f \text{ fresh future} \qquad \sigma'_\alpha = \mathit{Copy\&Merge}(\sigma_\alpha, \iota' ; \sigma_\alpha, \iota'')}{\alpha\big[F; \mathcal{R}_c[\iota.m_j(\iota')]; R; \sigma_\alpha\big] \,|||\, Q \longrightarrow \alpha\big[F; \mathcal{R}_c[f]; R::[m_j; \iota''; f]; \sigma'_\alpha\big] \,|||\, Q}$$

$$\text{SELFREPLY}\quad \frac{\sigma_\alpha(\iota)=f \qquad \sigma'_\alpha = \mathit{Copy\&Merge}(\sigma_\alpha, \iota_f ; \sigma_\alpha, \iota) \qquad \{f \mapsto \iota_f\} \in F}{\alpha\big[F; C; R; \sigma_\alpha\big] \,|||\, Q \longrightarrow \alpha\big[F; C; R; \sigma'_\alpha\big] \,|||\, Q}$$

Table 3: Parallel reduction (used or modified values are non-gray)

{Seq|Par}Schedule: method set × future set × future set × request queue → request × request queue

Given a set of method names to be served (the parameter of the *Serve* primitive), the set of futures calculated by the current thread, and the set of futures calculated by the other threads of the activity, those functions decide whether it is possible to serve a request sequentially or in parallel. The last parameter is the request queue that will be split into a request to be served and the remaining of the request queue. If no request can be served neither sequentially nor in parallel, both functions are undefined. We define $Futures(C)$ as the set of futures being computed by the current requests C. Then, parallel reduction $\longrightarrow$ is described in Table 2. We will denote by $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$. Table 2 consists of seven rules:

LOCAL triggers a local reduction $\rightarrow_{\text{loc}}$ described in Table 1.

ACTIVE creates a new activity: from an object and all its dependencies, this rule creates a new activity at a fresh location γ. The method m_0 is called at creation, the initial request is associated with $f_\varnothing$, a future that is never referenced and never used.

REQUEST invokes a request on a remote active object: if a current reduction point of activity α makes an invocation on another activity β this creates a fresh future f, and enqueues a new request in β. The parameter is deep copied to the destination's store.

ENDSERVICE finishes the service of a request: it adds an entry corresponding to the newly calculated result in the future value association list. The result object is copied to prevent further mutations.

REPLY sends a future value: if an activity α has a reference to a future f, and another activity β has a value associated to this future, the reference is replaced by the calculated value.

SERVE serves a new request sequentially: it relies on a call to *SeqSchedule* that returns a request $[m, f, \iota]$ and the remaining of the request queue R'. The request $[m, f, \iota]$ is served *by the current thread*. The *Serve* instruction is replaced by an empty object that will be stored, so that execution of the request can continue with the next instruction.
Note that *SeqSchedule* receives, the set of method names M, the set of futures of the current thread, the set of futures of the other threads, and the request queue.

PARSERVE serves a new request in parallel: it is similar to the preceding rule except that it relies on a call to *ParSchedule*, and that a new thread is created that will handle the new request to be served $[m, f, \iota]$.

Note that the previous rules do not consider the cases where the request or the reply source and destination is the same activity, two additional rules can be added for those cases (see Table 3).

We can prove that the reduction rule does not create inconsistent configurations with references to non-existing activities or futures.

Property 1 (Well formed reduction) $\longrightarrow$ *does not create references to futures or activities that do not*

$$\frac{\begin{array}{c}\forall f \in F'.\, \mathit{compatible}(f_j, f) \qquad \exists f \in F.\, \neg\mathit{compatible}(f_j, f) \\ m_j \in M \qquad \forall k < j.\, m_k \in M \Rightarrow \big(\mathit{compatible}(f_j, f_k) \ \wedge\ \exists f \in F'.\, \neg\mathit{compatible}(f_k, f)\big)\end{array}}{\mathit{SeqSchedule}(M, F, F', [m_i, f_i, \iota_i]^{i\in 1..N}) = ([m_j, f_j, \iota_j], [m_i, f_i, \iota_i]^{i\in 1..j-1} :: [m_i, f_i, \iota_i]^{i\in j+1..N})}$$

$$\frac{\begin{array}{c}\forall f \in F'.\, \mathit{compatible}(f_j, f) \qquad \forall f \in F.\, \mathit{compatible}(f_j, f) \\ m_j \in M \qquad \forall k < j.\, m_k \in M \Rightarrow \big(\mathit{compatible}(f_j, f_k) \ \wedge\ \exists f \in F'.\, \neg\mathit{compatible}(f_k, f)\big)\end{array}}{\mathit{ParSchedule}(M, F, F', [m_i, f_i, \iota_i]^{i\in 1..N}) = ([m_j, f_j, \iota_j], [m_i, f_i, \iota_i]^{i\in 1..j-1} :: [m_i, f_i, \iota_i]^{i\in j+1..N})}$$

Table 4: A possible definition of *SeqSchedule* and *ParSchedule*

exist; suppose $Q_0 \longrightarrow^* Q$ *and* $Q = \alpha\big[F; C; R; \sigma\big] \,|||\, Q'$ *then:*

$$\begin{array}{l}\sigma(\iota) = \beta \Rightarrow \exists F'\, C'\, R'\, \sigma'\, Q''.\, Q = \beta\big[F'; C'; R'; \sigma'\big] \,|||\, Q'' \quad \textit{and} \\ \sigma(\iota) = f \Rightarrow \exists \beta\, F'\, C'\, R'\, \sigma'\, Q''.\, Q = \beta\big[F'; C'; R'; \sigma'\big] \,|||\, Q'' \wedge \\ \big(\exists \iota.\{f \mapsto \iota\} \in F' \vee f \in \mathit{Futures}(C') \vee \exists m\, \iota.\, [m; f; \iota] \in R'\big)\end{array}$$

This property is quite easy to prove, let us consider the future case: each created future corresponds to an entry in the request queue of another activity, this future will then be treated as a current request, and later it will be stored indefinitely in the future value list.

5.3 Scheduling Requests

Several strategies could be designed for scheduling parallel or sequential services. Future identifiers can be used to identify requests uniquely; also it is easy to associate some meta-information with them (e.g. the name or the parameters of the invoked method). Consequently, we rely on a compatibility relation between future identifiers: $\mathit{compatible}(f, f')$ is true if requests corresponding to f and f' are compatible. We suppose this relation is symmetric.

Table 4 shows a suggested definition of functions *SeqSchedule* and *ParSchedule* which *maximises parallelism while ensuring that no two incompatible methods can be run in parallel*. The following of this section explains in what sense this definition is correct and optimal. The principle of the compatibility relation is that two requests served by two different threads should be compatible:

Property 2 (Compatibility) *If two requests are served by two different threads then they are compatible: suppose* $Q_0 \longrightarrow^* \alpha\big[F; C; R; \sigma\big] \,|||\, Q$ *then:*

$$C = [a_i \mapsto f_i]^{i\in 1..n} \| [a'_j \mapsto f'_j]^{j\in 1..m} \| C' \Rightarrow \forall i \in 1..n.\, \forall j \in 1..m.\, \mathit{compatible}(f_i, f'_j)$$

The non-trivial cases in the proof of this property are the two last rules of Table 2. The functions *SeqSchedule* and *ParSchedule* are defined such that they maintain this property: For SERVE, the served request (future f) should be compatible with the request of the threads that do not perform the *Serve* (futures *Futures(C')*); for PARSERVE, the request should also be compatible with the current thread (futures $\{f_i\}^{i\in 0..n}$).

We consider that parallelism is "maximised" if a new request is served whenever possible, and those services are performed by as many threads as possible. Thus, to maximise parallelism, a request should be served by the thread that performs the *Serve* operation only if there is an incompatible request served in that thread. The "maximal parallelism" property can then be formalised as an invariant:

Property 3 (Maximum parallelism) *Except the leftmost request of a thread, each request is incompatible with one of the other requests served by the same thread (and that precede it). More formally, if* $Q_0 \longrightarrow^* \alpha\big[F; C; R; \sigma\big] \,|||\, Q$ *then:*

$$C = [a_i \mapsto f_i]^{i \in 1..n} \| C' \wedge n > 1 \Rightarrow \forall i \in 2..n.\, \exists i' < i.\, \neg \mathit{compatible}(f_i, f_{i'})$$

This invariant is maintained because the leftmost request of a thread is the only one to progress, the set of futures currently served in one thread only grows when *SeqSchedule* returns a request, and decreases when a request finishes (the most recent request of the thread is removed).

The properties above justify the two first premises of the rules in Table 4. The last premise decides which request to serve. For this, we filter the request queue by the set M of method labels. Then we serve the first request that is compatible with all requests served by the other threads. These definitions ensure that any two requests served inside two different threads are always compatible, and also that requests are served in the order of the request queue filtered by the set of methods M, except that *a request can always overtake requests with which it is compatible*. We say that parallelism is maximised because we eagerly serve new requests on as many parallel threads as possible.

FIFO request service A classical service policy for active objects consists in serving all the requests in a first-come-first-serve order, called FIFO service. In the case of a FIFO service, only the service method m_0 performs *Serve* operations and m_0 simply consists of a loop: *while* (true) *Serve*(M_A) where M_A is the set of all method names[5]. m_0 is compatible with all the other methods and thus all services are parallel services (SERVE never occurs), but the number of parallel services depends on the compatibility relationship between requests.

A FIFO request service performs no filtering, at any time the first request that is compatible with all the requests currently served and all the requests preceding it in the queue is served. Consider the CAN example; inside a CAN peer, suppose there are two concurrent *add* operations running and there are two requests in the request queue: *join* and *monitor* (member of the group `monitoring`). We are not able to start the first request until the two *adds* finish. However, the second request (*monitor*) is compatible with all the others, and can be safely executed before *join*, concurrently with *adds*. For example, if we just considered the first request in the queue and would only serve it if it was compatible with the currently served requests, we would serve less requests in parallel.

Note on scheduling policy and compatible requests We conclude this section by explaining why it is reasonable to let a request overtake compatible ones. Allowing requests to overtake other compatible requests is, in fact, a logical choice. Whenever two compatible requests are served in parallel, compatibility implies that the operations of these requests can be freely interleaved. Overtaking is just a special case of interleaving, namely when all operations of one request are executed before the operations of the other request. Thus, even if requests were served in the exact incoming order, a request may actually overtake a single compatible one. Generalising the above remark, we can safely say that a request may overtake any number of compatible requests.

Consider the following example: suppose one request f_1 is being treated, and two requests are waiting $\boxed{f_2}\boxed{f_3}$; suppose $\mathit{compatible}(f_1, f_3)$, $\neg\mathit{compatible}(f_1, f_2)$, and $\mathit{compatible}(f_2, f_3)$. With a FIFO service policy and our scheduling, f_3 can be served before f_2, and f_2 will be served when f_1 terminates. Anyway, even if f_1 was not currently treated, f_2 and f_3 would run concurrently.

More generally, if all requests are necessarily served at some point in time, then allowing a request to be overtaken by compatible ones is a safe assumption. Otherwise, the order in which compatible requests are served becomes significant; indeed, a service policy could stop and serve only one of the two compatible requests. But since not serving all the requests received by an active object is considered as a mistake in the request serving policy, it is a safe to consider that *a request can always be overtaken by a compatible request*.

[5] *while* and *true* can be defined in pure ASP [7]

5.4 Properties

Property 4 (Compatibility) *If two requests are served by two different threads then they are compatible, on the contrary, two requests served by the same thread are incompatible; more formally suppose* $Q_0 \longrightarrow^* \alpha\big[F; C; R; \sigma\big] \,|||\, Q$ *then:*

$$\left.\begin{array}{l} C = [a_i \mapsto f_i]^{i\in 1..n} \| [a'_j \mapsto f'_j]^{j\in 1..m} \| C' \\ \wedge i \in 1..n \wedge j \in 1..m \end{array}\right\} \Rightarrow \mathit{compatible}(f_i, f'_j)$$

$$\left.\begin{array}{l} C = [a_i \mapsto f_i]^{i\in 1..n} \| C' \\ \wedge i \in 1..n \end{array}\right\} \Rightarrow i = 1 \vee \exists i' < i.\, \neg \mathit{compatible}(f_i, f_{i'})$$

This property is true because: 1) a request served inside a thread must be compatible with the futures of the other threads (first condition in Table 4); 2) a newly served request compatible with all the futures of the requests currently treated is served in a different thread; 3) a request incompatible with one of the futures of the current thread is served in the same thread; 4) the last request of a thread is the only one to progress, it finishes before the others, thus when one request finishes the set of futures currently served in the thread is the same as before the request was served, this ensures that the second assertion is maintained when a request finishes.

5.5 Race-conditions

Race-conditions on memory access are very difficult to handle and to reason about, especially with modern memory models like the one of Java [16], thus data race-conditions inside each activity should be avoided. This could be addressed by reasoning on ownership and separation [24], but also by identifying the part of the store manipulated by a deep copy by using techniques like ownership types [9]. In MultiASP, conflicts between local reading and writing operations can arise both with the local rules, but also with other rules reading the store: in ACTIVE, REQUEST, and ENDSERVICE, an object of the store is deep copied, meaning it is read together with all its dependencies. Note that REPLY cannot raise conflicts because it sends a value situated in an isolated part of the store. These are the only conflicts involving memory, but other race-conditions can appear in MultiASP. Race-conditions not related to memory access are described below.

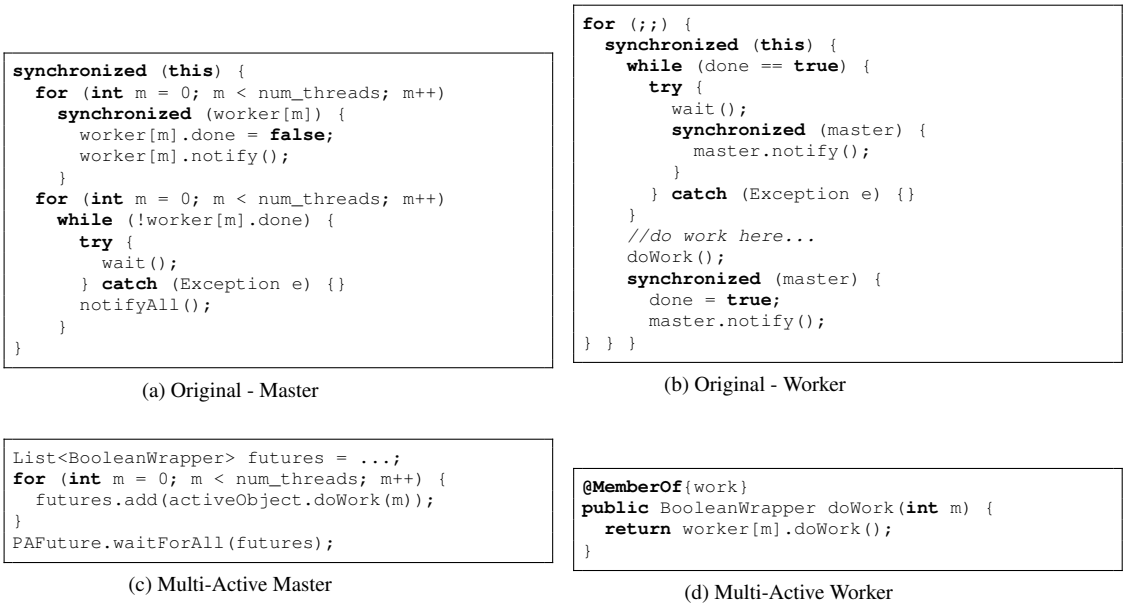
First, in ASP, the only source of non-determinism is the concurrent sending of requests toward the same activity. In MultiASP, concurrent sending of requests is also the major-source of non-determinism, but additionally two threads of the same activity can communicate towards the same activity. If the two invoked requests are non-compatible then this can influence the result of the computation.

Second, in MultiASP non-determinism can also originate from concurrent request service by two different threads: if one wants to ensure absence of race-condition between request services, then two services in two compatible methods should not try to serve the same method. More precisely, if *compatible*(f_1, f_2), and the service of f_1 performs a *Serve*(M_1), and the service of f_2 performs a *Serve*(M_2); then one should have $M_1 \cap M_2 = \varnothing$. If this condition is not met, then the two *Serve* operations act concurrently on the request queue leading potentially to very different results. Note that this constraint does not apply to FIFO request service.

The preceding remarks should be useful to decide how to annotate method compatibility to ensure absence of race-conditions, and, if desired, allow the active object to behave deterministically.

6 Evaluation

In this section we show that our proposal provides an effective compromise between programming simplicity and execution efficiency of parallel and distributed applications. After a short note on implementation, we compare the reference implementation of the NAS Parallel Benchmarks [3] (NPB) to a



```
synchronized (this) {
  for (int m = 0; m < num_threads; m++)
    synchronized (worker[m]) {
      worker[m].done = false;
      worker[m].notify();
    }
  for (int m = 0; m < num_threads; m++)
    while (!worker[m].done) {
      try {
        wait();
      } catch (Exception e) {}
      notifyAll();
    }
}
```

(a) Original - Master

```
for (;;) {
  synchronized (this) {
    while (done == true) {
      try {
        wait();
        synchronized (master) {
          master.notify();
        }
      } catch (Exception e) {}
    }
    //do work here...
    doWork();
    synchronized (master) {
      done = true;
      master.notify();
} } }
```

(b) Original - Worker

```
List<BooleanWrapper> futures = ...;
for (int m = 0; m < num_threads; m++) {
  futures.add(activeObject.doWork(m));
}
PAFuture.waitForAll(futures);
```

(c) Multi-Active Master

```
@MemberOf{work}
public BooleanWrapper doWork(int m) {
  return worker[m].doWork();
}
```

(d) Multi-Active Worker

Figure 8: NPB master-worker paradigm original and multi-active version

	BT	CG	FT	IS	LU	MG	SP
Lines (original / modified)	122 / 51	26 / 9	51 / 22	23 / 10	125 / 88	93 / 46	139 / 61
Percent of original	45.54%	34.62%	43.14%	43.48%	70.40%	49.46%	43.88%

Table 5: Lines of code dealing with parallelism: original vs. modified version of NPB

multi-active object version. We show that multi-active object concepts and annotations greatly simplify writing parallel code without significant impact on the application performance. Finally, we re-use our illustrative example to show how, with a few parallelism annotations, we transformed an active object implementation of the CAN [25] into a multi-active version that benefits from local parallelism, and thus is much more efficient. The purpose of this section is to show that our programming model is able to achieve the same performance as classical concurrent approach while simplifying the programming of distributed applications.

6.1 Implementation

Our proposal is implemented on top of ProActive [5], a Java middleware for parallel and distributed computing. Built on top of the Java standard APIs, mainly the Reflection API, it does not require any modification to the standard Java environment. No preprocessing or a modified compiler are required, all decisions are made at runtime by reifying constructors and method invocation on objects using a Meta-Object Procotol [22]. The flexibility and portability we obtain using these techniques can induce a slight overhead. However, as will be shown in the next section, it has no significant impact at the application level.

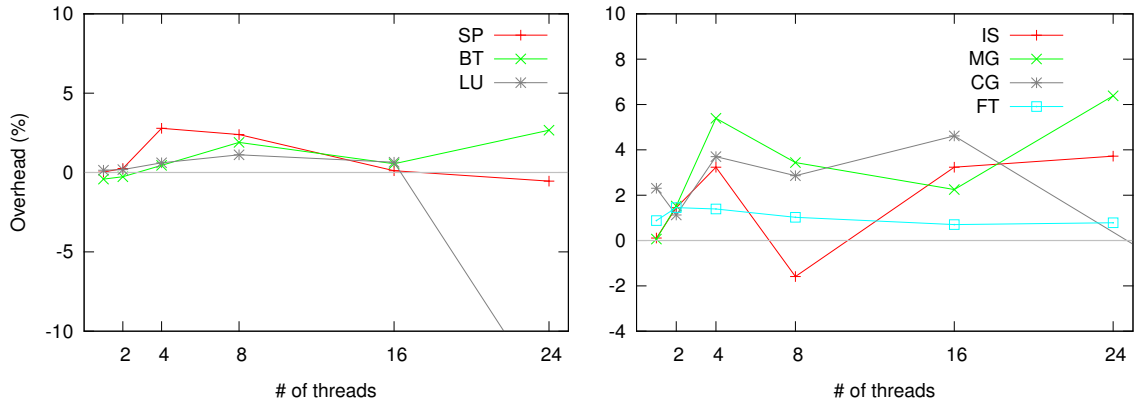


Figure 9: NAS Parallel Benchmark results – overhead of the multi-active version

6.2 NAS Parallel Benchmarks

When discussing new ways to parallelise code, it is important to check that applications written using the proposed model will not be running significantly slower than already existing solutions. We compare multi-active objects with Java threads based on a well-known parallel benchmark suite: the NAS Parallel Benchmarks [3]. To achieve a multi-active implementation, we modify the Java-based version of the benchmark [13] and create a multi-active object from each kernel. Our goal is to show that by using multi-active objects instead of standard Java threads and synchronized blocks, the code will become easier to write and to maintain, while the performance remains similar.

6.2.1 Simpler sources

Programming with Java threads may be a complicated task. In cases when simple *synchronized* blocks are not enough and thread synchronisation has to be managed explicitly, each *notify* and *wait* becomes a possible source of mistakes. The multi-active object paradigm on the other hand, offers a convenient way of parallelisation – with which similar results can be achieved, even by programmers not experts in parallel programming. In Figure 8 we show how the master-worker paradigm used in the NPB benchmarks can be expressed using multi-active objects. A single multi-active object replaces all worker threads, and futures and wait-by-necessity replace *wait()* and *notify()*. Overall, this makes the code easier to understand and to maintain.

To obtain a quantitative measure of "code simplicity", we count the number of lines of code needed to create and manage parallelism. We thus isolated all lines related to parallelism and counted them with JavaNCSS[6] for both versions of the benchmarks. Table 5 shows that a significant amount of code dealing with parallelism can be spared using multi-active objects.

6.2.2 Performance Overhead

To show that using multi-active has no major impact on the application performance, we have run the NPB kernels with problem size C, repeating each experiment five times and averaging the values. We ran our benchmarks on nodes of the PacaGrid[7] cluster, which are equipped with four hexa-core Intel Xeon

[6] `http://javancss.codehaus.org/`
[7] `http://proactive.inria.fr/pacagrid/`

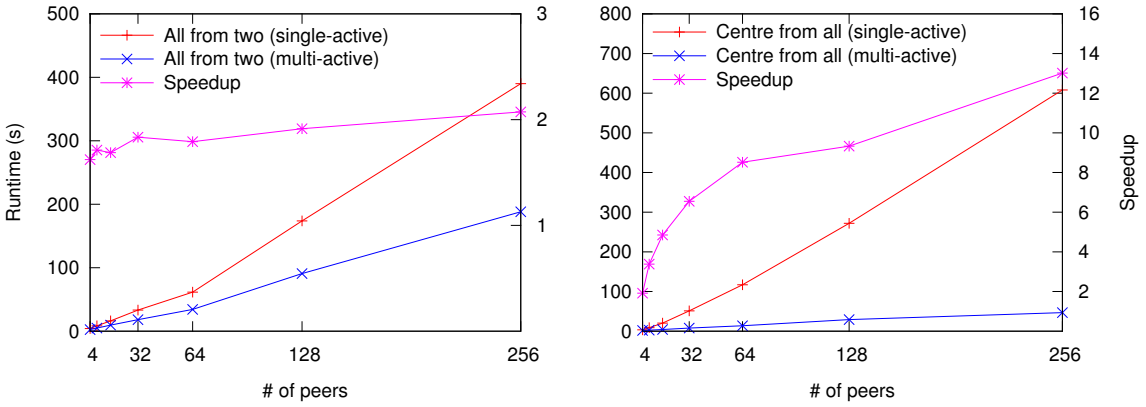


Figure 10: CAN experimental results

E7450 processors and 64GB of RAM. Kernels IS, MG, CG and FT were run using a Java 6 Hotspot 64-Bit Server VM. Kernels SP, BT and LU were run on a Java 7 Hotspot 64-Bit Server VM with the `-XX:-DontCompileHugeMethods` JIT option. This was necessary because the main computational methods have more than 8k bytecode instructions, and by default, such methods are never marked for compilation, leading to poor performance.

As displayed in Figure 9, the performance of the modified application is comparable to the original. The empirical standard deviation for the original and multi-active versions were measured and found to be almost identical. The difference in performance on LU with 24 threads could be explained by the fact that the original version creates 5 times 24 threads but only 24 are active at the same time, whereas the multi-active version only creates 24 threads.

These experiments show that multi-active objects can significantly reduce the amount of code needed for thread management without any significant performance hit. The implicit synchronisation of multi-active objects is almost as efficient as an explicit one.

6.3 CAN Experimental Results

When describing our model, we used the CAN peer-to-peer application as an example use-case for multi-active objects. We implemented the CAN example to illustrate the interplay between distribution and local parallelism allowed by multi-active objects, and show their efficiency in a distributed multicore environment. We created and populated a CAN using `join` and `add` operations presented in this paper, then we measured the benefits of `lookup` request parallelisation in the following situations:

- "All from two" – In this scenario, illustrated in Figure 3, we added an equal number of key-value pairs to all the peers in the network. We then used two peers, located at opposite corners of the CAN overlay to lookup all those values. Each corner sends `lookup` requests one after the other (a new request is only issued when the previous one has been enqueued). However, the results are all awaited at the end thanks to the use of futures. This experiment gives an insight about the overall throughput of the overlay.
- "Centre from all" – In this test case, illustrated in Figure 4, all the peers lookup concurrently a key situated in a peer at the centre of the CAN. This experiment highlights the scalability of a peer under heavy load.

Each experiment consisted in performing 50 times one scenario and measuring the overall execution time. Dividing the network into equal parts and assigning keys to peers is considered as part of the setup and is not measured. We measure the time necessary for triggering all the lookups and retrieving the values, the difference of execution time between successive runs is negligible. We used up to 128 machines located at the Sophia-Antipolis site of the Grid5000[8] platform. All hosts are interconnected through Gigabit Ethernet, and are equipped with quad-core CPUs (Intel Xeon E5520, AMD Opteron 275 and AMD Opteron 2218). All experiments were run using the Java 7 Hotspot 64-Bit Server VM. Each value amounted to 24KB.

Figure 10 shows the execution times and speedup for several sizes for the two scenarios. Both scenarios achieve significant speedup, however, the gain in the first scenario is smaller because the lookups are issued from the two corners in sequence; the sequential sending of the initial lookups limits the quantity of parallel lookups present at the same time in the network. In the second case, the active object version has a bottleneck because the center peer can only reply to one request at a time whereas those requests can be highly parallelised with our model.

As shown before, these speedups are achieved by just adding a few simple annotations to the class declaration without changing any of the implemented logic.

7 Comparison with Related Works

PAM Parallel actor monitors [27] (PAM) provide multi-threading capacities to actors based on explicit scheduling functions. However, we believe that the compatibility annotations of multi-active objects provide a higher level of abstraction than PAM, and that this high-level of abstraction is what makes active-objects and actors easy to program.

Cooperative Multi-threading The main differences between our approach and active-objects with cooperative multi-threading like JCoBox [26] and Creol [20] are the following. First, Creol-like languages are not really multi-threaded, thus they do not necessarily address the issue of efficiency on multicore architectures; JCoBox proposes a shared immutable state that can be used efficiently on multicore architectures but as the distributed implementation is still a prototype, it is difficult to study how an application mixing local concurrency and distribution like our CAN example would behave. Second, concerning synchronisation, in cooperative multi-threaded solutions between explicit release points (`await`s) the programs are executed sequentially. Adequately placing those release points is the main challenge in programming in Creol or JCoBox: too many release points leads to a complex interleaving between sequential code portions, whereas not enough of them will probably lead to a deadlock. This issue was illustrated in Figure 2 using the JCoBox notation.

Multi-active object provide an alternative approach to local concurrency in active objects: annotations allow the programmer to reason at the level of compatibility rules which is high-level and allows parallelism to be expressed in a simple manner. Then compatible methods are run concurrently, with potential race-conditions but also local multi-threading.

Overall, this paper provides a programming model where distribution and concurrency is much more transparent that in JCoBox and Creol; this difference in point of view explains most of the differences between the two models: transparent vs. explicit futures, compatibility annotations vs. explicit thread release. However, the principles of multi-active objects could be applied to an active object language with explicit futures and explicit release points, but in this case thread activation (after an await statement) should take into account compatibility information.

[8] `http://www.grid5000.fr/`

```
public class Peer {
...
/** @when !runningAdd
*/
public JoinResponse join(Peer other) {
  runningJoin = true;
  //do logic
  runningJoin = false;
}
private boolean inSplittingZone(Key k){ ... }

/** @compatible add(Key, Serializable), join(Peer)
    @when !runningJoin || !inSplittingZone(k)
*/
public void add(Key k, Serializable value) {
  runningAdd = true;
  // do logic
  runningAdd = false;
}}
```

Figure 11: The CAN Peer annotated in JAC

An annotation-based framework for parallel computing Concerning annotations for concurrency, Cunha and Sobral [10] use Java annotations to parallelise sequential objects in an OpenMP fashion. A method can be called asynchronously if it is flagged with the `Future` annotation, but the programmer must follow the flow of futures carefully and declare which methods can access them. There is also an `ActiveObject` annotation that creates a proxy and a scheduler, but its exact semantics is not well-defined in [10]. In our opinion, JAC's and our compatibility rules offer a greater control and a higher abstraction level than OpenMP style fork-join blocks, they are also better adapted to active-objects.

JAC's annotations Our annotation systems looks very much like JAC's proposal, and this paper could also be seen as an adaptation of a concurrency model *à la* JAC to the active object model. The inheritance model of JAC annotations is well designed and resembles the way our annotations are inherited from class to class. However, multi-active objects offer a simpler annotation system and a higher synchronisation logic encapsulation for the programmer. Moreover, the dynamic compatibility rules of multi-active objects are not directly translatable into JAC annotations. Figure 11 illustrates how one would like to translate into JAC the CAN example with dynamic compatibility (Figure 6). In practice however, this code does not work because the setting and reading of flags can happen concurrently, and JAC does not synchronise on these condition expressions. A solution would be to encapsulate these expressions in synchronized methods, but JAC does not accept methods for preconditions either. Compared to JAC, we think multi-active objects are simpler to program, have stronger properties, and are more suited to distribution. In particular, the transparent inclusion of annotations into an active object model leads to our mind to a powerful and interesting programming model.

X10 X10 [8] is a programming language that adopts a fairly new model, called partitioned global address space (PGAS). In this model, computations are performed in multiple places (possibly on various computational units) simultaneously. Data in one place is accessible remotely, and is not movable once created. Computations inside places are locally synchronous, but inter-place activities are asynchronous. This decouples places and ensures global parallelism. While this model seems fundamentally different from active-objects, the possibilities provided by the two are comparable. Places can host multiple activities, resulting in a similar service to what multi-active objects offer. One thing that the language does not ensure, is encapsulation, which even though might be costly in terms of performance, eases the development of applications [29]. We try in this work to allow for both powerful local concurrency, and strong

encapsulation of each (multi-)active object.

8 Conclusion

In this paper we have presented the *multi-active object* model, which is a novel approach for parallelising execution inside active objects. Our purpose is to find solutions to two of the main issues with active objects: inefficiency on modern multi-core architectures, and potential deadlocking upon re-entrant calls.

In this paper we presented the multi-active object programming model and its operational semantics. Active objects have been extended with annotations to allow and control multi-threaded execution inside them. The annotations can be written from a high-level point of view by declaring compatibility relations between the different concerns an active object manages. Relying on those annotations, request services can be scheduled such that parallelism is maximised while preventing two incompatible requests from being served in parallel. The semantics of the new programming model have been precisely and formally defined in Section 5. This formalisation allowed us to precisely identify the sources of race-conditions between concerns. We proved that our scheduling policy is safe, that is, it ensures that only compatible requests can execute in different threads.

We implemented the proposed model in Java, starting from an existing active object library, and ran several experiments to ensure that our runtime is stable and efficient. The two experiments discussed in this paper were the NAS Parallel Benchmark, and a peer-to-peer network application. The results of the NAS Parallel Benchmark show that the performance of our multi-active object implementation is similar to the manually multi-threaded version. At the same time we showed that code dedicated to parallelism is much simpler when using multi-active objects. Finally, the peer-to-peer experiment illustrated the performance gain brought by multi-active objects compared to a classical active object version. Overall, we showed that multi-active objects outperform simple active objects, and are easier to program than classical multi-threading.

The main originality of our contribution relies on the interplay between formal definitions that allow the precise study of the language properties and a practical middleware implementation efficient enough to compete with classical multi-threading benchmarks. We spent a considerable effort to design a practical programming model. This led us to the definition of dynamic constraints for compatibility allowing a fine-grain control over local concurrency. Overall we think the annotation system we provide is particularly adapted to and well-integrated with the active-object model.

References

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[3] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991.

[4] Frank S. De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*. Springer, 2007.

[5] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[6] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.

[7] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. *Information and Computation*, 207(4):459–495, 2009.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Conference Proceedings of OOPSLA'05*. ACM, 2005.

[9] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] CA Cunha and JL Sobral. An annotation-based framework for parallel computing. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 113–120. IEEE Computer Society, 2007.

[11] Frank S. de Boer, Mario Bravetti, Immo Grabe, Matias Lee, Martin Steffen, and Gianluigi Zavattaro. A petri net based analysis of deadlocks for active objects and futures. In *FACS*, Lecture Notes in Computer Science. Springer, 2012.

[12] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. *ECOOP 2006–Object-Oriented Programming*, 2006.

[13] M. Frumkin, M. Schultz, H. Jin, and J. Yan. Implementation of NAS parallel benchmarks in Java. In *a Poster session at ACM 2000 Java Grande Conference*, 2000.

[14] Elena Giachino and Cosimo Laneve. Analysis of deadlocks in object groups. In Roberto Bruni and Jürgen Dingel, editors, *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2011.

[15] A. Gordon, P. Hankin, and S. Lassen. Compilation and equivalence of imperative objects. In *Foundations of Software Technology and Theoretical Computer Science*, pages 74–87. Springer, 1997.

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[17] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[18] M. Haustein and K.P. Lohr. Jac: declarative Java concurrency. *Concurrency and Computation: Practice and Experience*, 18(5):519–546, 2006.

[19] Ludovic Henrio, Fabrice Huet, Zsolt István, and Gheorghen Sebestyén. Adapting active objects to multicore architectures. In *ISPDC*. IEEE Computer Society, 2011.

[20] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, November 2006.

[21] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[22] Gregor Kiczales and Jim des Rivieres. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[23] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[24] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375:271–307, April 2007.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM, 2001.

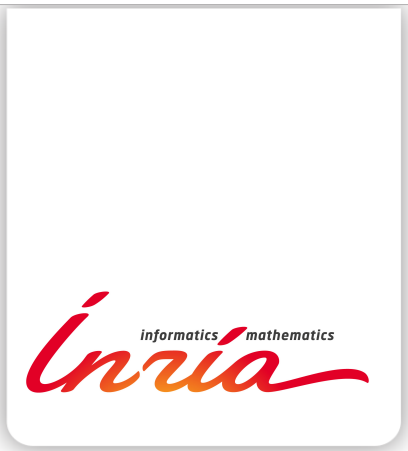
[26] J. Schafer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming*, pages 275–299, 2010.

[27] C. Scholliers, É. Tanter, and W. De Meuter. Parallel actor monitors. In *14th Brazilian Symposium on Programming Languages*, 2010.

[28] A. Shanneb, J. Potter, and J. Noble. Exclusion requirements and potential concurrency for composite objects. *Science of Computer Programming*, 58(3):344–365, 2005.

[29] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, page 45. ACM, 1986.

[30] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming ABCL/1. In *Conference proceedings of OOPLSA '86*, NY, USA, 1986. ACM.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399