Declarative Scheduling for Active Objects

Ludovic Henrio, Justine Rochas

Declarative Scheduling for Active Objects

Ludovic Henrio Justine Rochas

Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271
06900 Sophia Antipolis, France
ludovic.henrio@cnrs.fr, justine.rochas@unice.fr

ABSTRACT

Active objects are programming constructs that abstract distribution and help to handle concurrency. In this paper, we extend the multiactive object programming model to offer a priority specification mechanism. This mechanism allows programmers to have control on the scheduling of requests. The priority representation is based on a dependency graph which makes it very convenient to use. This article shows how to use this mechanism from the programmer side, and exposes the main properties of the dependency graph. The software architecture of our implementation is also presented, as it can be applied to various scheduling systems. Finally, we validate our approach through a microbenchmark that shows that the overhead of our priority representation is rather low. On the whole, we provide a general pattern to introduce a prioritized scheduling in active objects or in any other concurrent systems. The resulting framework is shown to be fine-grained, user-friendly, and efficient.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.2 [**Language Classification**]: Concurrent, Distributed, and Parallel Languages

General Terms

Languages, Design

Keywords

concurrency, active objects, priority scheduling, annotations

1. INTRODUCTION

The active object programming model [10] aims to facilitate the writing of distributed applications. It mainly provides asynchronous remote method calls and mechanisms to prevent data races inside an active object. Several implementations of active objects came out with different ways of handling concurrency and execution of requests. Some implementations, like Creol [9], offer release points that can be used to make requests progress in an interleaved manner. Then, this concept has been extended to groups of objects instead of objects, such as in JCoBox [13] or in ABS [8]. Other implementations like ASP [3] enforce a strict sequential execution of requests. to prevent interleaved executions.

However, all these models do not take advantage of multicore architectures as they are intrinsically mono-threaded. To overcome this problem and to increase the applicability of active-object oriented programming, recent extensions of existing models have been designed. In the context of actors [2], Parallel Actor Monitors [14] introduce schedulers that are pluggable to an actor[2, 4], and that can manage parallel execution of requests. Also, an extension of the active object model, called multiactive objects [7], enables local parallelism at the request level without giving up simplicity and safety provided by active objects: programmers can declare which requests can be run in parallel through a customized specification language. In this paper, we enhance the coordination capabilities of the multiactive object programming model by introducing a simple way to specify priority of execution within a multiactive object, still in a declarative fashion. Programmers are thus able to assign priorities to requests, in order to influence on the internal scheduling of multiactive objects. For that, we introduce a mini metalanguage that relies on annotations. Internally, we represent the priorities using a dependency graph, a structure that is well adapted to represent a partial order, and that is well suited for priorities. Our annotation-based language is easy to program and intuitive; still it gives a fine-grain control on the request scheduling, and this paper also shows a microbenchmark proving its efficiency.

The paper is organized as follows. Section 2 presents the background, especially the multiactive object programming model. Section 3 introduces our graph-based priority specification mechanism with its main properties. Section 4 exposes the general software architecture of our implementation. Section 5 shows a microbenchmark that evaluates the performance of the priority graph representation. Section 6 compares our work with existing mechanisms which can provide scheduling controls in active objects.

2. MULTIACTIVE OBJECTS

The multiactive object programming model [7] enables request parallelism within an active object: several requests can progress at the same time. This is different from the cooperative not preemptive scheduling offered by some active object

SAC'14 March 24-28, 2014, Gyeongju, Korea.

```
@DefineGroups({
  @Group(name="group1", selfCompatible=true),
  @Group(name="group2", selfCompatible=false)
})
@DefineRules({
  @Compatible({"group1", "group2"})
})
public class MyClass {
  @MemberOf("group1")
  public ... method1(...) { ... }
  @MemberOf("group2")
  public ... method2(...) { ... }
  ...
}
```

Figure 1: Example of annotated class

languages [8, 9] because with those, only one request is allowed to progress at a time, to prevent data race-conditions. Oppositely, the goal of multiactive objects is to take advantage of multicore architectures. Multiactive objects aim at keeping the benefits of active objects in terms of ease of programming and safety of execution, while introducing a true parallelism in a simple and controlled way. For that, they rely on a meta language in order to specify which requests can safely be executed concurrently. Programmers can use this language to define requests that are *compatible* for concurrent execution. In practice, methods of a multiactive object have to be partitioned into *groups* and then, compatibility can be specified for those groups. Groups are meant to provide a coarser granularity than requests to express execution compatibility. The way to form groups is left to programmers: one can choose to group methods that are semantically related while another would rather keep the number of groups the smallest possible.

The specific language we propose has been developed using the Java annotation mechanism. Annotations are processed at runtime to decide whether a request can be executed in parallel with others, request compatibility could even be decided at runtime (e.g. depending on request parameters). Figure 1 shows how we can annotate a regular Java class to allow an active object to process several requests in parallel. In this example, we use firstly the `@DefineGroups` annotation to define two groups, named `group1` and `group2`. Secondly, the `@DefineRules` annotation specifies which groups are compatible together, i.e. which requests are allowed to be run in parallel with which others. Thirdly, in the class body, `method1` is assigned to `group1` thanks to the `@MemberOf` annotation. The same pattern is applied to `method2`. A few lines of such annotations enables significant speedup at the application level, thanks to multi-processing of requests.

To efficiently execute requests according to the defined compatibilities, multiactive objects enforce an adapted *First In First Out* policy with possibility to overtake. More precisely, a request that is waiting in the reception queue is executed if it is compatible with:

- All requests that are currently executing, *and*
- All requests that are before in the reception queue.

Any request that satisfies those two conditions is said to be *ready to execute*; it can be immediately executed in a new thread. Note that, if we execute the second request before the first one, then this second request will not prevent the first one from executing since they are compatible. Also, to avoid killing the whole performance by creating too many threads on the fly, the number of threads that run at the same time can be limited, again through an annotation. In this case, multiactive objects rely on a fixed thread pool to execute requests. As a consequence of this limit, requests that are ready to execute might wait if all available threads are busy. Those requests that are ready to execute but that cannot be executed because of lack of threads form a new queue that we call the *ready queue*, in opposition with the *reception queue* that contains requests that are not ready to be served (for incompatibility reasons).This mechanism is pictured on Figure 2. Improving the scheduling of requests in the ready queue is necessary to optimize request executions, and adapt it to the needs of programmers.

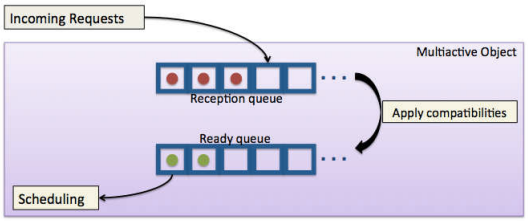

Figure 2: The internal queues of a multiactive object

3. GRAPH-BASED PRIORITIES

3.1 Presentation

Multiactive objects enable multi-threading within an active object. While this model is convenient for running multiple requests in parallel, it does not enable a particular scheduling other than *First Compatible First Out*, as explained in Section 2. Defining a *priority* relationship between requests would greatly improve the efficiency by reducing the response time of important requests. We define a priority relationship as the fact that requests having a high priority can overtake requests which have a low priority. To implement this scheduling mechanism, we must address two questions: "how can the programmer specify the priorities?" and "how to internally represent the priorities?"

To specify the priorities, we propose dedicated annotations that benefit from the simplicity of a declarative mechanism, and are consistent with the existing programming model. To internally represent the priorities, we have developed a model based on a dependency graph. The motivation of this representation is that it can express a partial order, whereas a representation based on integers to represent priority would result in a total ordering of requests. Another major problem problem with the integer-based approach is that the programmer is bothered with the internal representation of priorities, having to take into account all the values previously assigned before defining a new one. Instead, we propose to specify the priorities by establishing a dependency between request groups. The internal representation of priorities is not exposed to the user, making the annotations higher level, and therefore, easier to use. Internally, we maintain a graph representation where nodes represent groups and directed edges represent the priority relation.

The new annotations we have developed are defined as follows. First, the `@DefinePriorities` annotation contains several chains of dependencies, each of them defined in a `@PriorityOrder` annotation. A `@PriorityOrder` annotation contains a sequence of sets of groups, each defined in a `@Set` annotation. A `@Set` annotation can contain several request groups that have the same position in the graph; the next set in the sequence has a lower priority than the previous one.

```
@DefinePriorities({
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G2"}),
    @Set(groupNames = {"G4"}),
    @Set(groupNames = {"G5"})
  }),
  @PriorityOrder({
    @Set(groupNames = {"G1"}),
    @Set(groupNames = {"G3"}),
    @Set(groupNames = {"G5"})
  })
})
```

Figure 3: Example of priority declaration

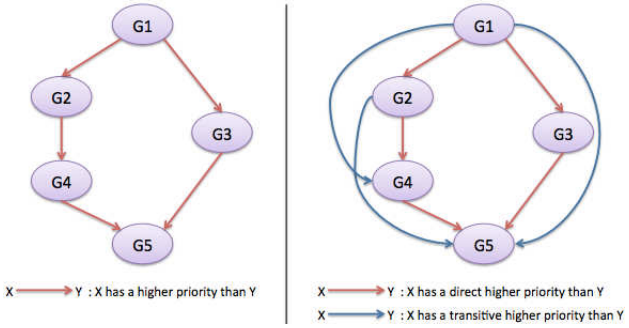

Figure 4: Graph of Figure 3 and its transitive closure

An example of priority declaration is given on Figure 3, and the corresponding graph is depicted on Figure 4. In this graph, we prioritize five groups of requests. In particular, we specify that `G1` has the highest priority and that `G5` has the lowest priority. We can notice in this graph that group `G3` is not related with group `G2` nor group `G4`. Indeed, those groups do not belong to the same dependency tree, as can be seen on Figure 4. Note that there are several ways to define the same dependency graph. This model is very flexible because it is easy to add new priorities, independently from the priorities previously defined.

3.2 Scheduling policy and properties

When using priorities, the scheduling policy applied in multiactive objects depends on one main information: Can a request overtake another one when it is inserted in the ready queue. This information lies on the graph itself. In this section, we first introduce the main properties of the graph to be able to express the insertion process afterwards.

Construction. We ensure that the dependency graph contains no cycle by construction. When a dependency is processed, if it introduces a cycle in the graph, we do not add the dependence; the execution is not stopped but an error message outputs. Also, in the graph, each group belongs to a single node. When a dependency is added, we first check whether a node labeled with this request group is already in the graph. If such a node exists, we only add edges to the graph, else we also add a node for the new group.

Notations. Here are the notations we use in the following.

- A request is generally denoted R, a group G and the ready queue Q.
- We start from a dependency graph defining a priority relation between groups, denoted with the relation $\longrightarrow$, whose operands are nodes of the graph.
- In the graph, reachability is denoted $\longrightarrow^+$. $G_1 \longrightarrow^+ G_2$ means that there exists a directed path from the group G_1 to the group G_2. $\longrightarrow^+$ is the *transitive closure* of $\longrightarrow$. In other words, $G_1 \longrightarrow^+ G_2$ means that group G_1 has a higher priority than group G_2.

Overtakability. The main information we want to extract from the graph is whether a request can overtake another one. This information lies in the dependency graph.

- A request of group G_1 *can overtake* a request of group G_2 if and only if $G_1 \longrightarrow^+ G_2$, i.e. if there is a directed path from G_1 to G_2.
- A request of group G_1 has no priority relation with a request of group G_2 if $\neg\ G_1 \longrightarrow^+ G_2 \ \wedge\ \neg\ G_2 \longrightarrow^+ G_1$, i.e. if there is no directed path from G_1 to G_2 and no directed path from G_2 to G_1; this is denoted $G_1 // G_2$. Note that, for any group G, $G // G$.

Consequently, G_1 *cannot overtake* G_2 if either $G_1 // G_2$ or $G_2 \longrightarrow^+ G_1$.

Insertion. When a new request is inserted in the ready queue, overtakability properties are used to determine the position of the new request. An incoming request of group G must be inserted in the ready queue just before the first request whose group can be overtaken by the group G. More precisely, considering an incoming request R belonging to group G, and a ready queue made of $R_1, R_2, ..., R_n$ belonging to groups $G_1, G_2, ..., G_n$, R is inserted just before the smallest R_i such that $G \longrightarrow^+ G_i$, or at the end of the queue if no such R_i exists. More formally, we define the insertion process as follows:

DEFINITION 1 (INSERTION PROCESS). *Suppose* $group(R) = G$, $Q = [R_1, ..., R_n]$, *and* $\forall i \in 1..n$, $group(R_i) = G_i$
if $\forall i,\ G_i \longrightarrow^+ G \ \vee\ G // G_i$, *then* $insert(R, Q) = [R_1, ...R_n, R]$
else let $j = min(i \mid G \longrightarrow^+ G_i)$ *in*
$insert(R, Q) = [R_1, ..., R_{j-1}, R, R_j, ..., R_n]$

Considering this insertion process, the ready queue is always ordered according to the overtakability relationship:

PROPERTY 1 (READY QUEUE ORDERING). *The ready queue is always ordered such that if* $i \leq j$, *then* $G_i \longrightarrow^+ G_j$ *or* $G_i // G_j$

We prove this property by checking that it is maintained both when a request is inserted by the insertion process, and when the first request is removed from the ready queue.

To sum up, the reordering of requests in the ready queue relies on the possibility that one request overtakes another one. This knowledge is provided by the dependency group, which is expressed by simple annotations giving a priority order between request groups. Potentially, at each request insertion, the whole graph must be explored for each request in the ready queue. Although our approach is very expressive, the graph exploration during the insertion process might lead to a performance problem if the graph is large; this strongly depends on the way we internally represent the graph. Section 5 will finally measure the performance of our solutions.

3.3 Internal representation enhancement

To address the potential performance problem, we have developed an enhanced version of the graph-based priority representation that allows us to quicken the insertion process relatively to a naive graph exploration. From the priority graph, we compute its *transitive closure*. This graph of transitive closure can be seen as a binary matrix that stores all the possible combinations of overtakability. In this matrix M, of size NxN where N is the number of nodes, we store a

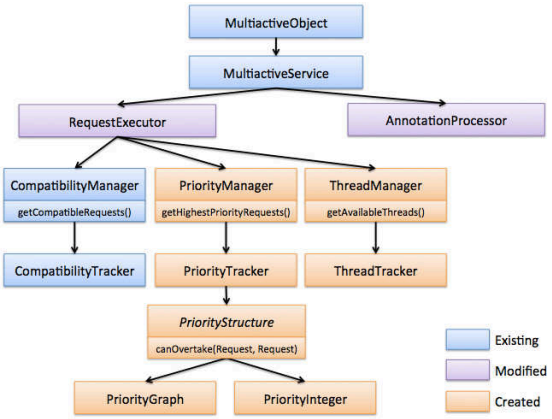

Figure 5: Multiactive object classes with scheduling

positive value in $M[n_1][n_2]$ if there exists an edge from n_1 to n_2 in the transitive closure of the priority graph. To speed up the insertion process, we access the matrix entries instead of exploring the graph. Both the priority graph and the matrix is built once, when the annotations are processed, and used the whole execution time.

The only drawback of the matrix representation, compared to the graph structure, is the memory used. There are exactly N^2 entries in the matrix instead of N nodes for the graph plus $N \times (N+1)/2$ edges maximum (most of the time much less). This can be a problem if there exists a lot of groups since the graph must be kept in memory the whole execution time. In practice, the number of methods of an active objects that are remotely invoked (and subject to priority ordering) is rather small because of the cost/benefit of communication time in a distributed environment. Still, if there is actually a lot of groups, one can choose to turn off the matrix optimization and use the plain graph. A better representation of the matrix could also be used. Choosing the right representation is indeed a trade off between time performance and memory space; however, we never had to make such a choice in our first experiments.

4. SOFTWARE ARCHITECTURE

This section describes the overall architecture we implemented to schedule requests according to priorities in the multiactive object framework. It illustrates how we implement a request selection and how we apply a scheduling mechanism based on code annotations from the programmer. A multiactive object embeds an object of type `MultiactiveService`. The `MultiactiveService` has two purposes. First, it relies on an `AnnotationProcessor` that reads annotations and initializes all the structures required by a multiactive object; these structures store compatibilities, priorities, and threads. Second, the `MultiactiveService` has a `RequestExecutor` that schedules requests according to compatibilities, priorities, and available threads. For that, the `RequestExecutor` relies on managers, that are `CompatibilityManager`, `PriorityManager`, and `ThreadManager`. The manager classes can access the structures that store the information processed from the annotations; they also use dynamic structures (the trackers). Figure 5 shows an UML diagram describing the class composition of multiactive objects and the most important interfaces.

On one hand, the `RequestExecutor` has access to the manager interfaces. On the other hand, each entity manages its own state internally. More precisely, to execute requests, the `RequestExecutor` first filters incoming requests using the `CompatibilityManager`. Among ready requests, a second filter is applied using the `PriorityManager`, that sends back a list of requests that have the highest priority. Then, the final decision regarding the execution of a request depends on the `ThreadManager`, that knows if some threads are available. After applying those three filters sequentially, remaining requests are the ones that are actually executed.

The `PriorityTracker` contains the ready queue; it adheres to the Producer/Consumer design pattern. On one hand, a dedicated thread registers compatible requests to the ready queue. On the other hand, another dedicated thread polls the ready queue to retrieve highest priority requests. This synchronization is implemented using Java concurrency mechanisms, and is totally hidden from programmers. When a request is registered to the ready queue, the `PriorityStructure` interface is queried to know where the request must be inserted. The concrete `PriorityStructure` used for that can be either our dependency graph (`PriorityGraph`), or other priority representations if needed (for example here, `PriorityInteger`). A concrete `PriorityStructure` must implement the `canOvertake(request1, request2)` method that says whether a request can be executed before another one. In the `PriorityGraph` case, either the `canOvertake` method goes through the graph to retrieve this information, or it simply checks the overtakability matrix if this feature is enabled.

This design can easily be adapted to other implementations of active objects, provided that the language they rely on supports annotation processing and reflexivity.

5. EXPERIMENTAL EVALUATION

5.1 Environment

In this section, we present a microbenchmark evaluating the general overhead of priorities. It is run on a single machine because, even though the objective of active objects is to ease distributed programming, our priority mechanism applies at the level of a single active object. Priorities contribute to the efficiency of local parallelism, not to the efficiency of distributed execution.

The machine used to run our experiments has four 4-core Intel Core Q6600 processors and 8GB of memory. Experiments are run using a Java 7 virtual machine. We developed our experiment using the EventCloud platform [12] because it provides an API to easily run multiactive objects. The EventCloud platform relies on the multiactive version of the ProActive middleware [1]. We have modified ProActive to introduce our scheduling mechanisms; we have then used this customized version as a basis of EventCloud. The general process of the microbenchmark is the following. We first create and run a single multiactive object. Then, we send requests to it and we record relevant metrics using logging mechanisms.

5.2 Overhead of priorities

We show here a test case where the graph of priorities is not trivial. Our objective is to evaluate the average insertion time of a request in the ready queue, i.e. the overhead of priorities. On top of the test class, we define ten groups using the `@DefinePriorities` annotation, from G1 to G10. Each of these groups has a single belonging request, from g1 to g10. All groups are declared compatible, otherwise, they

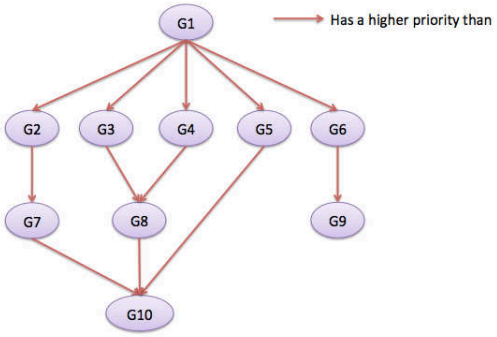

Figure 6: Dependency graph used for the microbenchmark

would never be in the ready queue at the same time, and we could never apply priorities. The priority dependencies between them are declared using priority annotations; the corresponding graph is expanded on Figure 6.

We perform two runs with those annotations: one using the plain dependency graph, and one using the reachability matrix. Also, to evaluate the performance of both priority representations compared to simpler approaches, we perform a third run with a third priority representation that we have implemented. It is based on integer ordering: instead of building a graph, we simply assign an integer to a group to represent its priority. Note that doing so is more restrictive than expanding the priorities with a graph: the integer ordering is total whereas the graph produces a partial order[1].

In our test case, we sequentially send a request of each group in a predefined order. The chosen order is the following (with no particular meaning): g7, g1, g2, g9, g4, g10, g8, g3, g6, g5. Using a predefined sequence of requests allows us to have deterministic results while experimenting with all parts of the graph. We send 500 requests per group, so in total, 5000 requests are sent to the multiactive object. This test case follows a worst-case scenario in two senses. First, all requests have the smallest possible body, made at most of two instructions: a logging instruction and a return instruction. In practice, requests sent to a multiactive object should be long enough to be balanced with communication time. As a consequence, in practice, the performance of multiactive objects should be higher than in our experimental situation. Second, we intentionally block the execution until all requests we want to experiment are received and put in the ready queue. This way, we end up having a big ready queue, which again makes the scenario the worst possible. In brief, this microbenchmark tests the priority specification mechanism at its limits, and the performance can only be better in real cases.

As a first observation, to measure the overhead of the priority mechanism, we compare it with the service time of an empty request. Even with the worst priority representation, the time to insert a request using the dependency graph is always below 10% of the minimal service time of a request when the ready queue contains less than 200 requests. This means that, considering the fact that the service time is the time we would pay anyway even if there was no priorities, managing the structures and all the mechanisms introduced for priorities is not costly compared to a priorityless approach.

We evaluate the overhead that we pay in general when using the graph representation compared to the integer representation. We do not look specifically into the overhead of a given priority, but rather at the global overhead. For that, we average insertion times every 50 consecutive requests sent from the same group. Thus, we have 10 measurements per group, one for each stage of consecutive requests. We finally average the metrics of all groups for each stage. The results, displayed on Figure 7, correspond to the average insertion time of a request of any group. First of all, if we consider the run performed with the plain dependency graph and the run performed with the integer ordering, we can see that, as the queue gets bigger, the time to find the right position in the queue increases much more in the graph case. This is due to the fact that, at each request considered, we need to go through the graph to find (i) the group of the request to insert and (ii) the group of the request to consider, to know if the request to insert can overtake the considered request. Even if the overhead due to the priority mechanism is quite small, it is still more than twice higher than the overhead of the integer representation. However, thanks to the matrix representation, we managed to reach the same performance as the integer representation, because we reduce the overtakability problem to a fundamental operation (accessing a box of the matrix), as with the integer representation. It is thus possible to benefit from the ease of programming of the graph representation without requiring more time than the most efficient priority representation.

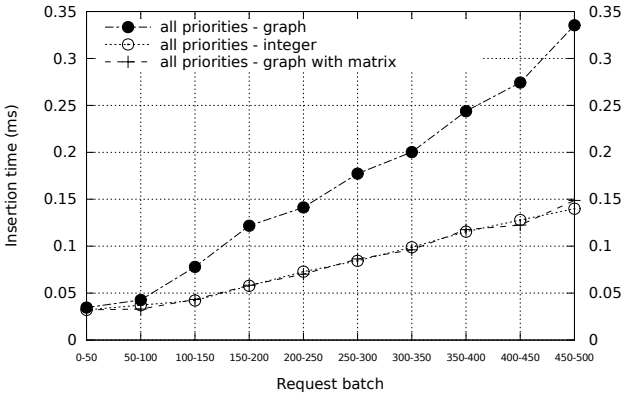

Figure 7: Average insertion time of requests

6. RELATED WORKS

To provide control on the scheduling of requests in active objects, one classical approach is to design the active objects with a default scheduling policy and to propose to programmers to plug their own schedulers on top of it. In PAM [14], which is quite close to multiactive objects, a customized scheduler can be defined by the programmer and then plugged to any actor to control the scheduling of requests within it. In particular, if programmers want an actor to respond to a priority policy, the policy must be directly implemented in the scheduler definition, i.e. the entire priority queue must be implemented. In a similar way, ABS [8], which offers an active object implementation based on groups of objects, named *cogs*, proposes to write customized schedulers and to plug them to active objects. However, this way to control the scheduling of requests is fairly complex, as it requires programmers to write a significant part of the scheduling code, and to directly manipulate requests in the queue. Consequently, the expertise expected from programmers is high.

[1]We use one of the linear extensions of the graph of Figure 6

It is worth noting that, contrarily to what happens in actor programming [15, 5], our objective is not to allow an active object to treat thousands of messages per second because each message in our framework represents a communication between two remote machines. Also, we execute pure Java programs where each logical thread is a Java thread; thus a multiactive object is not meant to run thousands of threads in parallel, as context switching would be too costly.

JAC [6] is an extension of Java that decouples concurrency constructs from the application logic. Similarly to us, it uses annotations to deal with local concurrency but outside the context of active objects or actors. JAC offers a set of annotations to specify method compatibility for concurrent execution. JAC also offers a `@schedule` annotation, placed on top of a method, in which programmers can define an additional method that will be called just before the initial method call, to decide whether the initial method should actually be executed. The additional method has access to the list of waiting requests, which makes it possible to define fine-grained scheduling policies. Yet, accessing the list of waiting requests is quite low level.

Oppositely to the schedulers cited above, we introduce here a high level priority specification mechanism only based on a small set of declarative instructions, which is still very expressive. Programmers can thus focus on the design of their business code, and on the compatibility and priority relationship between requests. Our approach requires much less knowledge on the internal structure of active objects and requests than existing works.

Finally, the authors of [11] extended the possibilities of the Creol [9] active object language to be able to prioritize execution of requests in a declarative fashion. This work is the closest to ours, request priorities can be defined both on the server side (upon a method definition) and on the client side (when calling the method). To decide the final priority of a particular request, the active object applies a deterministic function that takes into account both the method call priority and the method definition priority. Although this work introduce scheduling controls at a high level, it is still complex to use because programmers do not know right away the priority value of a request. In our approach, we chose to assign priorities on the server side only, to simplify the process and to avoid over-use of priorities on the client side. Our priorities are also highly connected to compatibilities, and compatibility is not covered by Creol as it does not support multi-threading. Finally, the priority relation in Creol is based on integer ordering, whereas our priority definition is graph-based, and thus allows partial ordering.

7. CONCLUSION

In this paper, we presented a way to control the scheduling within active objects by introducing a declarative priority specification mechanism. The priorities allow programmers to control request scheduling within multiactive objects. As the mechanism relies on annotations, it is easy to assign priorities without having to code the intended behavior of the scheduler directly in the business code. Related works on high-level scheduling do not provide such an abstract specification, or were fairly complex to use and required programmers to know the internal mechanisms of active objects. In our work, we succeeded in providing a high level language to specify priority of execution, without asking programmers to directly manipulate requests. Our work is adapted to multiactive objects, the first active object model supporting both local multi-threading and large-scale distributed execution.

The priority specification mechanism we have developed relies on a dependency graph: when a group has a higher priority than another one, a dependency is created between them. This specification is easy to use and expressive, but requires time and processing resources. We showed that, provided a reachability matrix is used instead of a simple priority graph, we reach the same performance as a representation based on integer ordering, while the graph-based representation is more expressive. Consequently, we elaborated a priority specification mechanism for active objects that is user-friendly, expressive, and also efficient.

8. REFERENCES

[1] The Proactive middleware. `proactive.inria.fr`.

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

[3] D. Caromel, L. Henrio, and B. Serpette. Asynchronous sequential processes. *Information and Computation*, 2009.

[4] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. ECOOP'06. Springer-Verlag.

[5] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

[6] M. Haustein and K.-P. Löhr. Jac: declarative java concurrency. *Concurrency and Computation: Practice and Experience*, 2006.

[7] L. Henrio, F. Huet, and Z. István. Multi-threaded active objects. In C. Julien and R. De Nicola, editors, *COORDINATION'13*, LNCS. Springer, June 2013.

[8] E. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. Abs: A core language for abstract behavioral specification. In B. Aichernig, F. Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects*, LNCS. Springer Berlin Heidelberg, 2012.

[9] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 2006.

[10] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[11] B. Nobakht, F. S. de Boer, M. M. Jaghoori, and R. Schlatte. Programming and deployment of active objects with application-level scheduling. SAC '12. ACM, 2012.

[12] L. Pellegrino, F. Baude, and I. Alshabani. Towards a scalable cloud-based rdf storage offering a pub/sub query service. In *CLOUD COMPUTING 2012*.

[13] J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *ECOOP 2010*, LNCS. Springer Berlin Heidelberg.

[14] C. Scholliers, É. Tanter, and W. De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Science of Computer Programming*, 2013.

[15] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In J. Vitek, editor, *ECOOP 2008*, volume 5142 of *LNCS*. Springer.